ASM-One

Professionelles Assembler-Entwicklungssystem

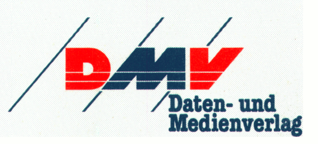
Für Commodore Amiga



Bücher

Zeitschriften

Software



ASM-ONE

Version 1.01

Assembler
Debugger
Monitor
Editor

für

Commodore Amiga

DMV-Verlag • Postfach 250 • 3440 Eschwege

Herzlich willkommen bei ASM-ONE, einem professionellen Assembler-Paket für Commodore Amiga, das Ihnen neben einem schnellen Assembler einen professionellen Editor, einen Monitor zur Disassemblierung und einen Debugger mit Step-, Watch- und Jump-Funktionen bietet.

Wir weisen Sie darauf hin, daß Verlag und Autoren weder juristische Verantwortung noch Haftung für Schäden und Datenverluste übernehmen können, die durch den Gebrauch des Programms entstanden sind.

Wir sind Ihnen für Verbesserungsvorschläge und Hinweise auf Programmfehler über unsere Hotline dankbar.

Programm und Handbuch: Rune Gram-Madsen

Übersetzung: Holger Lubitz

Computer-Satz: Andreas und Dagmar Ballhaus

Printed in Germany

ASM-ONE

Inhaltsverzeichnis

Vorwort

Jetzt, nachdem die Arbeit getan ist und Ihnen mit ASM-ONE ein Komplettpaket zur Programmierung des Commodore Amiga vorliegt, das mit integriertem Monitor und Debugger jeden Assembler-Programmierer zufriedenstellen wird, möchte ich mich bei allen Personen bedanken, die durch ihre Unterstützung und Ermutigung ASM-ONE möglich gemacht haben.

Mit ihren Tips und Vorschlägen haben Jesper Steen Møller von Danish Softech Aps, Sune Trudslev, Mads Henriksen und Michael Nielsen geholfen. Colin Fox (Pyramyd Designs) und Bruce Dawson (CygnusSoft Software) wird für die Programmierung und Freigabe der »Requester.Library« gedankt.

Besonderen Dank möchte ich meinen Eltern aussprechen, die ausreichendes Vertrauen in meine Fähigkeiten bewiesen haben, und ein Dankeschön auch an Henrik Brink, Florian Schroecker, Dan Jensen und Michael Holm für ihr Interesse an ASM-ONE.

Erwähnt sei auch mein Hund Jason, weil er mich bei den nächtlichen Arbeiten wachgehalten und morgens durch sein Bellen wieder geweckt hat.

Ihnen wünsche ich viel Erfolg mit ASM-ONE und viele nützliche und gute Assembler-Programme für Ihren Commodore Amiga.

Im Oktober 1990

Rune Gram-Madsen

1. Einführung

Willkommen bei ASM-ONE! Dieses Programmpaket ist eine integrierte Entwicklungsumgebung aus einem Assembler, einem Editor, einem Debugger und einem Monitor. Damit haben Sie alle notwendigen Funktionen in einem Paket integriert, was die Entwicklung neuer Programme sowohl kürzer als auch benutzerfreundlicher werden läßt. Sie können Programme jeder Art und jeden Umfangs schreiben, die einzige Beschränkung legt Ihnen dabei der verfügbare Speicherplatz auf.

Kurzer Überblick über den ASM-ONE-Makro-Assembler:

Integrierter Source-Level-Debugger: Er bietet die Möglichkeit, das Programm im Einzelschrittmodus abarbeiten zu lassen, während frei wählbare Speicherbereiche und alle Registerinhalte angezeigt, Breakpoints freigesetzt und sämtliche Register editiert werden können.

Kompatibilität mit ALink und BLink: Hierdurch werden Hochsprachen-Programmierer in die Lage versetzt, Assembler-Routinen in ihre Programme einzubinden.

Schnelle Assemblierung mit 50000 bis 70000 Befehlszeilen pro Minute.

ASM-ONE ist **Quellcode-kompatibel** mit allen bisherigen Assemblern.

Viele zusätzliche Direktiven erlauben benutzerfreundlichere Quellcode-Erstellung.

Integrierter superschneller Editor mit Befehlen zur Blockmanipulation, zum Suchen, zum Ersetzen, zum Markieren und mit Makrofähigkeit.

Echtzeit-Fullscreen-Monitor mit Disassemblierung, Hexadezimalausgabe, ASCII-Ausgabe, Sprungmöglichkeit und Adreßmarkierung.

ASM-ONE ist benutzerfreundlich durch **Menüs mit Tastaturkürzeln** für alle Funktionen.

ASM-ONE bietet alternativ auch **absolute Speicherbelegung**, wodurch der Assembler zum optimalen Entwicklungswerkzeug für Spiele und Demos wird.

Weiterhin Unterstützung **binärer Includes** mit »INCBIN« und »EXTERN«.

1.1 Möglichkeiten mit ASM-ONE

Die Programmierung in Assembler erfordert mehr Aufmerksamkeit auf winzige Details als in allen anderen Programmiersprachen. Andererseits ist es möglich, Assembler-Programme durch Ausnutzung der genau passenden Assembler-Befehle quasi maßzuschneidern, wodurch schnellere Programme möglich werden.

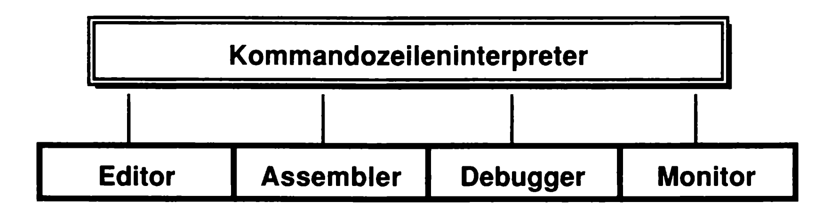
Da die Programmierung eine so zeitaufwendige Beschäftigung ist, ist es wichtig, daß die Entwicklungsumgebung so schnell und flexibel wie möglich ist. Das ist der Grund, warum in **ASM-ONE Assembler, Editor, Monitor** und **Debugger** integriert sind. Dadurch sparen Sie Zeit. Alle Funktionen können sowohl über Menüs als auch über die Tastatur angewählt werden, wobei die Tastaturbedienung sich anbietet, sobald Sie mit diesem Programm vertraut sind.

ASM-ONE ist sowohl zur Systemprogrammierung als auch zur Spieleprogrammierung besonders geeignet. Denn ASM-ONE ist **Makro-Assembler-kompatibel** mit Include-Files, verschiedenen Sektionen und so weiter und erlaubt auch direkte Speicherbelegung, Monitorfunktionen etc. für Systemprogrammierer. Dadurch werden beide Programmieraufgaben optimal unterstützt, ohne die jeweils andere auszuschließen.

1.2 Der ASM-ONE-Makro-Assembler

Um ASM-ONE richtig zu benutzen, ist das grundlegende Verständnis der Funktionsweise Voraussetzung.

Die Basis der Konzeption von ASM-ONE ist der Kommandozeileninterpreter, der in der Bedienung dem CLI vergleichbar ist. In ihn gelangen Sie automatisch, nachdem Sie Typ und Größe des von ASM-ONE zu belegenden Arbeitsspeichers gewählt haben. Von dort aus können Sie alle Modi von ASM-ONE anwählen und alle Befehle eingeben. ASM-ONE gliedert sich wie folgt:



Die Vorgehensweise beim Erstellen eines Programms mit ASM-ONE ist folgende:

1. **Schreiben** Sie einen **Programmtext,** oder laden Sie einen in den Editor.
2. **Assemblieren** Sie diesen sogenannten Quelltext, und kehren Sie bei Syntaxfehlern in den Editor zurück. (1)
3. Versuchen Sie, das **assemblierte Programm** zu **starten**. Falls Sie logische Fehler im Programm bemerken, korrigieren Sie diese ebenfalls im Editor (1), oder wechseln Sie in den Debugger, wenn Sie die Fehler nicht auf Anhieb entdecken können.

Anmerkung: Bevor Sie ein neu geschriebenes Programm zum erstenmal starten, sollten Sie alles **abspeichern**, da beim Start eines Assembler-Programms die komplette Kontrolle über den Computer an dieses Programm übergeben wird und im Falle eines Absturzes der Quelltext im Editor und das assemblierte Programm verloren sind.

2. Installation von ASM-ONE

Die mitgelieferte **Programmdiskette ist bootfähig**. Sie können die Diskette also statt der Workbench-Diskette zum Starten Ihres Rechners benutzen und sofort mit ASM-ONE arbeiten. Nachdem Sie ASM-ONE ausprobiert haben, möchten Sie ASM-ONE vielleicht von der Workbench starten, einen Drucker benutzen oder ASM-ONE auf Ihrer Festplatte installieren.

Sie haben für die weitere Benutzung von ASM-ONE zwei Möglichkeiten, von denen Sie eine auswählen sollten:

1. **Direktes Booten von der Programmdiskette**
2. **Booten von der Workbench**

2.1 Installation für direktes Booten von der Programmdiskette

Falls Sie Ihren Drucker nach dem Booten von der ASM-ONE-Diskette nutzen wollen, müssen Sie die folgenden Dateien von Ihrer Workbench-Diskette auf die ASM-ONE-Diskette kopieren:

```
devs/serial.device              nach   devs/
devs/parallel.device            nach   devs/
devs/printer.device             nach   devs/
devs/system-configuration       nach   devs/
devs/printers/<ihr Drucker>     nach   devs/printers/
l/Port-Handler                  nach   l/
```

Das Kopieren dieser Dateien auf einem virusinfizierten Computer oder von einer virusinfizierten Diskette könnte zu unerwünschten Modifikationen der ASM-ONE-Diskette führen. Wenn Sie nicht sicher sind, daß Ihr Computer und Ihre Workbench-Diskette virusfrei sind, fahren Sie mit Abschnitt 2.2 fort.

Stellen Sie sicher, daß Sie denjenigen **Druckertreiber** kopieren, der in der Systemkonfiguration eingestellt ist. Sie können dies überprüfen, indem Sie »Preferences« starten. Schlagen Sie gegebenenfalls in Ihrem Amiga-Handbuch nach, wie Sie »Preferences« bedienen.

Der sicherste Weg ist, den gesamten Inhalt der »devs/« und »l/« Verzeichnisse auf die ASM-ONE-Bootdiskette zu kopieren. Zwar sind hierbei mehr Dateien zu kopieren, dafür haben Sie danach alle auf einer Diskette.

Wenn Sie sich nicht völlig sicher sind, wie Sie dies anstellen, sollten Sie zu Abschnitt 2.2 übergehen.

2.2 Installation für das Booten von der Workbench

Dies ist viel einfacher als die unter 2.1 beschriebene Methode, weil Sie hierzu nur eine einzige Datei von der ASM-ONE-Bootdiskette auf Ihre Workbench-Diskette kopieren müssen:

`libs/req.library` **nach** **`libs/`**

Nun können Sie mit Ihrer Workbench-Diskette wie gewöhnlich booten, danach die ASM-ONE-Diskette einlegen und das ASM-ONE-Icon anklicken.

3. Die ersten Schritte

Nach dem Laden von ASM-ONE erscheint folgende **Eingabeaufforderung**:

```
ALLOCATE Fast/Chip/Abs>
```

Wenn Sie ein Fast-Memory haben, geben Sie nun »F« ein, andernfalls »C« und bestätigen mit »Return«. Nun werden Sie gefragt, wieviel Speicher Sie für den Programmtext und das assemblierte Programm zur Verfügung stellen möchten.

```
WORKSPACE (max.???) Kb>
```

Geben Sie jetzt die gewünschte **Speichergröße** ein, und bestätigen Sie mit »Return«. 100 KByte sind für die Beispiele auf der ASM-ONE-Diskette ausreichend. Sie befinden sich nun im Kommandozeilen-Interpreter und sehen den Eingabeprompt

```
>
```

Nun ist ASM-ONE aktiv und bereit zur Arbeit. Ein guter Einstieg ist, wenn Sie zum Kennenlernen die Datei

»GettingStarted.S«

laden. Wählen Sie im Menü »Project« den Menüpunkt »Read-File« an, und ein großes Datei-Requester wird geöffnet. Klicken Sie einmal auf das Examples-Verzeichnis und dann zweimal auf die Datei »GettingStarted.S«.

Ist die Datei geladen, wählen Sie zum Ansehen die **Funktion »Edit«** im **Menü »Assembler«** an. Sie können nun alle Funktionen im Menü »Edit Funct.« benutzen. Den Editor verlassen Sie mit der Taste »Esc« oder durch die Funktion »Exit« im Menü »Edit Funct.«.

Von der **Kommandozeile** aus assemblieren Sie den Quellcode des Beispiels »GettingStarted.S«, indem Sie

```
>a
```

eingeben und mit »Return« bestätigen.

Die Antwort wird

```
Pass1..
Pass2..
No Errors
>
```

sein. Geben Sie nun

```
>J
```

ein, und bestätigen Sie mit »Return«, um das assemblierte Beispielprogramm zu starten, das Sie mit »Leertaste« wieder beenden können.

Um ASM-ONE richtig kennenzulernen, sollten Sie auch die übrigen **Beispielprogramme** im Verzeichnis »Examples« assemblieren. Dabei schnuppern Sie ruhig auch einmal durch die verschiedenen Menüs von ASM-ONE, und wählen Sie Editor, Debugger oder Monitor im Menü »Assembler« probehalber an.

4. Einstiegskapitel

Wenn Sie schon etwas über Maschinensprache wissen sollten, können Sie dieses Kapitel überspringen. Es ist als grundlegende Einführung in Maschinensprache und in die verschiedenen Zahlensysteme aufgebaut.

4.1 Warum Maschinensprache?

Maschinensprache ist die einzige Sprache, die der MC68000-Prozessor des Amiga versteht. Auch wenn Sie Programme in Basic oder C verfassen, werden diese zur Ausführung in Maschinensprache übersetzt. In Basic wird ein Programm Zeile für Zeile übersetzt, was entsprechend lange dauert. Ein C-Programm wird zwar durch einen Compiler an einem Stück in Maschinensprache übersetzt, aber da jeder C-Befehl in mehrere Maschinensprache-Befehle übersetzt werden muß und diese Befehle je nach Güte des verwendeten Compilers nicht immer in der günstigsten Weise aufeinander folgen, ist der erzeugte Code nicht optimal. Absolut optimalen Code erreicht man nur durch die direkte Programmierung in Maschinensprache.

Und noch ein weiterer Vorteil von Maschinensprache ist zu nennen: Wenn Sie wissen wollen, wie Ihr Rechner arbeitet und wie zum Beispiel die verschiedenen Komponenten der Hardware zusammenarbeiten, dann ist Maschinensprache ein guter Weg, um zu diesem Verständnis zu gelangen. In Maschinensprache können Sie jeden Teil Ihres Rechners genau steuern. Daher werden nicht nur die meisten Spiele direkt in Maschinensprache geschrieben, sondern auch viele Utilities und Anwendungsprogramme und natürlich auch ASM-ONE.

Auch Programmierer, die normalerweise mit Hochsprachen arbeiten, müssen öfters Teile des Codes in Maschinensprache einbinden, um in ihrer Sprache unlösbare Probleme zu bewältigen oder um die Geschwindigkeit zeitkritischer Routinen zu erhöhen.

Ich persönlich habe Maschinensprache immer allen anderen Sprachen vorgezogen, weil ich es mag, die komplette Kontrolle über die Maschine zu haben, und zudem schnelle Programme sehr schätze.

4.2 Das Amiga-System

Der Amiga basiert im Prinzip auf dem gleichen Konzept wie alle Computer: Er besitzt einen Prozessor (MC 68000), einen Hauptspeicher (RAM, Random Access Memory) und einige Spezialchips. Diese Spezialchips sind es auch, die den Amiga berühmt gemacht haben, denn er bekam gleich drei davon in die Wiege gelegt: **Agnus** ermöglicht mit dem Blitter schnelle Animationen, **Denise** sorgt für Grafik mit bis zu 4096 Farben, und **Paula** produziert den vierstimmigen Stereo-Sound und kontrolliert Ports und Diskettenlaufwerke.

Das **Betriebssystem** des Amiga befindet sich im Nur-Lese-Speicher, auch ROM genannt. Es kann als ein Programm erklärt werden, das sich um nahezu alles kümmert, was im Computer abläuft. Das Betriebssystem lädt Programme, kontrolliert die Maus und bewegt den Mauszeiger zur richtigen Stelle. Der Bildschirm mit Texten und Grafiken wird ebenfalls vom Betriebssystem aufgebaut.

Das Betriebssystem belegt derzeit 256 KByte Speicherplatz. Exakt sind es 262144 Byte, was mit der **digitalen Architektur** eines Computers zusammenhängt, in der es nur die Schaltzustände »Ein« und »Aus« gibt, statt der gewohnten Ziffern 0

bis 9. Um 1000 Byte anzusprechen, braucht man in der digitalen Architektur zehn Adreßleitungen, mit denen sich

2^10 = 2*2*2*2*2*2*2*2*2*2 = 1024 Byte

ansprechen lassen. Im dezimalen Zahlensystem wären nur 1000 Byte möglich.

Der **Hauptspeicher** in einem Standard-Amiga mit 512 KByte ist daher exakt 512*1024 = 524288 Byte groß. Jedes Byte setzt sich aus 8 Bit zusammen, die nur die Zustände »Ein« oder »Aus« annehmen können. Daraus folgt, daß ein Byte 2^8 = 256 verschiedene Werte annehmen kann, also die Zahlen von 0 bis 255. Definiert man die Zahl als vorzeichenbehaftet und das oberste Bit als Vorzeichen-Bit, verschiebt sich der Wertebereich auf -128 bis +127.

4.3 Die verschiedenen Zahlensysteme

Haben Sie es bereits bemerkt? Sie wurden soeben mit einem neuen Zahlensystem bekanntgemacht. Es wurden Bits erwähnt, die nur die Werte »Ein« und »Aus« annehmen können, während im dezimalen Zahlensystem Ziffern von 0 bis 9 verwendet werden. Das **binäre Zahlensystem** unterscheidet sich vom dezimalen dadurch, daß es nur die Ziffern 0 und 1 kennt.

Nehmen wir zum Beispiel folgende Zahl, wobei das Prozentzeichen vor einer Zahl gewöhnlich eine Zahl im binären Zahlensystem markiert:

n = %100111

Um diesen Wert in eine Dezimalzahl umzuwandeln, können Sie den eingebauten Rechner von ASM-ONE nutzen. Geben Sie einfach

```
>?%100111
```

ein, und das Ergebnis sieht dann folgendermaßen aus:

Hex	Dezimal	ASCII	Binär
$00000027	39	"...'"	%00000000.00000000.00000000.00100111

Sie können sehen, daß »%100111« im Dezimalsystem dem Wert »39« entspricht.

Ein anderer Weg, die Zahl umzuwandeln, ist folgender: Eine normale Dezimalzahl wird aus Einsern, Zehnern, Hundertern usw. zusammengesetzt. Die Stellen einer Binärzahl entsprechen nicht Zehner-, sondern Zweierpotenzen, also 1, 2, 4 usw. Um unser Beispiel umzuwandeln, rechnet man also wie folgt:

% 1 0 0 1 1 1 = 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 1*1 = 39

Eine Umwandlung in die andere Richtung ist natürlich ebenso möglich:

39 =	**39/32**	**= 1*32**	**+ 7**
	7/16	**= 0*16**	**+ 7**
	7/8	**= 0*8**	**+ 7**
	7/4	**= 1*4**	**+ 3**
	3/2	**= 1*2**	**+ 1**
	1/1	**= 1*1**	**+ 0**
= %100111			

Ein großer Nachteil binärer Zahlen ist, daß sie viel länger und daher schwerer zu lesen sind als dezimale Zahlen. Wir müssen sie aber verwenden, weil Zahlen in einem Computer binär gespeichert werden.

Um binäre Zahlen aber einfacher lesbar zu machen, benutzt man häufig die **hexadezimale Notation**. In »Hex« können einzelne Ziffern Werte von 0 bis 15 annehmen, doch da die »Ziffern« 10 bis 15 im dezimalen Zahlensystem schon zweistellig wären, benutzt man statt dessen die Buchstaben A bis F wie folgt:

Dezimal Hexadezimal	Dezimal Hexadezimal	Dezimal Hexadezimal
0 = 0	6 = 6	12 = c
1 = 1	7 = 7	13 = d
2 = 2	8 = 8	14 = e
3 = 3	9 = 9	15 = f
4 = 4	10 = a	
5 = 5	11 = b	

Nehmen wir nochmals die binäre Zahl unseres Beispiels.

n = %00100111

Sie kann ziemlich einfach in das hexadezimale Zahlensystem umgewandelt werden:

0000 = 0	0100 = 4	1000 = 8	1100 = c
0001 = 1	0101 = 5	1001 = 9	1101 = d
0010 = 2	0110 = 6	1010 = a	1110 = e
0011 = 3	0111 = 7	1011 = b	1111 = f

Also entspricht »%00100111« = »$27«.

4.4 Die Speicherbelegung

Jedes einzelne Byte hat im Speicher seine eigene Adresse. Sie können zum Beispiel auf die Adresse »300000« zugreifen, die sich im Chip-Memory befindet, dem Speicherbereich, auf den die Spezialchips des Amiga zugreifen. Hier ist eine einfache Speicherbelegungstabelle des Amiga:

$000000-$07ffff	Chip-Memory
$080000-$1fffff	freier Adreßraum für zusätzliches Chip-Memory
$200000-$9fffff	freier Adreßraum für Fast-Memory
$a00000-$beffff	reserviert
$bfd000-$bfdf00	CIA B gerade Adressen
$bfe001-$bfef01	CIA A ungerade Adressen
$c00000-$c7ffff	Ranger-Memory (Speichererweiterung im A500)
$c80000-$dfefff	reserviert
$dff000-$dfffff	Register der Spezialchips
$e00000-$e7ffff	reserviert
$e80000-$efffff	Adreßraum für Autokonfiguration
$f00000-$f7ffff	Erweiterungs-ROMs
$f80000-$fbffff	Adreßraum für Kickstart-Erweit. auf 512 KByte
$fc0000-$ffffff	Kickstart-ROM (256 KByte)

Wie Sie vielleicht sehen konnten, läßt sich der Amiga auf bis zu 10 MByte RAM erweitern. Um jedoch an die zusätzlichen 1,5 MByte Chip-Memory heranzukommen, muß der Amiga mit dem »Super Fat Agnus 8372« ausgerüstet sein.

4.5 Das Programm

Um ein Programm im Speicher abzulegen, benutzt der Amiga den Hauptspeicher. Das Programm wird im Hauptspeicher als eine lange Liste von Zahlen abgelegt, einer Serie von binären Einsen und Nullen. Diese Werte sind für den Prozessor die Instruktionen. Die einzelnen **Befehle des 68000er** können dabei zwischen 2 und 10 Byte lang sein.

Eine der kürzesten Instruktionen ist »RTS« (Return from Subroutine, Rückkehr aus einem Unterprogramm) und hat den Wert %0100111001110101 im binären Zahlensystem, $4e75 hexadezimal und 20085 dezimal. Die meisten anderen Instruktionen sind komplizierter, weil viele von ihnen zusätzliche Operanden besitzen.

Da es schwer möglich ist, sich all diese Nummern zu merken, benutzt man zur Programmierung in Maschinensprache einen **Assembler**, der die Namen aller Befehle versteht. Wenn Sie also beispielsweise »RTS« schreiben, übersetzt der Assembler das automatisch in den oben erwähnten Wert $4e75.

Natürlich gibt es noch viel mehr Befehle, aber es ist schon möglich, ein kurzes Programm zu schreiben, ohne sie alle zu kennen.

Zwei neue Befehle neben »RTS« brauchen wir dennoch.

MOVE <Quelle>,<Ziel>
ADD <Quelle>,<Ziel>

Der Befehl »Move« überträgt einen Wert an eine andere Speicherstelle. Der Befehl »Add« addiert einen Wert zu dem einer anderen Speicherstelle.

Mit diesen Befehlen schreiben Sie nun Ihr erstes Assembler-Programm. Dazu wechseln Sie in den Editor, löschen zuerst

alles, was im Editor steht, mit dem Kommando »Zap Source« im Menü »Project« und drücken dann »Esc« . Ein einfaches Programm aus den drei vorgestellten Kommandos:

```
START   MOVE.L   #0,$000000       ; Lösche Adresse 0
                                  ; (32 bit)
        ADD.L    #$1234,$000000       ; Addiere
                                  ; $1234 hinzu
        RTS
```

Dies sollte das erste und einzige Mal bleiben, daß Sie in einem Programm den Inhalt der Adresse »0« modifizieren. Zwar wird sie vom System nicht genutzt, es ist aber nur zu wahrscheinlich, daß die Adresse »0« aufgrund von Progammierfehlern in anderen Programmen überschrieben wird. Es ist bei der Programmierung des Amiga sehr schlechter Stil, sich auf feste Speicheradressen zu verlassen. (Anmerkung des Übersetzers)

Nachdem Sie das Programm in den Editor eingegeben haben, verlassen Sie ihn durch nochmaligen Druck auf »Esc«. Versuchen Sie nun das Programm zu assemblieren, indem Sie

```
>A
```

eingeben. Das Ergebnis sollte folgendes sein:

```
Pass 1..
Pass 2..
No Errors
>
```

Falls Sie eine Fehlermeldung erhalten, müssen Sie erneut in den Editor wechseln, um den Fehler zu finden und zu korrigieren. Bei einem so einfachen Programm kann es sich eigentlich nur um einen Tippfehler handeln.

Wenn Sie das Programm assembliert haben, können Sie es starten, indem Sie

```
>J
```

eingeben. Geben Sie nun

```
>H.L 0
```

ein, um das Ergebnis zu sehen. Die erste Zeile der nun folgenden Ausgabe sollte ungefähr so aussehen:

00000000 00001234 00.....

Die erste Spalte gibt die Adresse an, an der Sie in den Speicher schauen. Der nächste Wert ist 00001234, der Inhalt der Adresse »0«.

Dieser Wert ist korrekt. Wenn Sie nun wieder unser Beispielprogramm betrachten, werden Sie sehen, daß die Adresse »00000000« mit »0« beschrieben wurde. Dann wurde der Wert $1234 zum Inhalt der Adresse »00000000« addiert. Als Endergebnis befindet sich $1234 in Adresse »00000000«.

4.6 Die Prozessorregister

Beim Schreiben eines Programms werden Sie sehr häufig Zwischenwerte in Berechnungen oder als Schleifenzähler benötigen. Um einen solchen Wert zu speichern, können Sie eine Adresse im Hauptspeicher benutzen. Aber wenn Sie diesen Wert häufig brauchen sollten, empfiehlt es sich, statt dessen ein Register zu verwenden, da dies schneller ist.

Sie können sich die Register wie Speicheradressen vorstellen, die jedoch innerhalb des Prozessors gehalten werden. Auf dem Motorola MC 68000 sind 15 verschiedene Register für die Programmierung verfügbar.

Diese Register gliedern sich in zwei Gruppen.

D0-D7	**Datenregister**	**32 Bit**
A0-A6	**Adreßregister**	**32 Bit**

Die Datenregister sind dabei die flexibleren: Sie können sowohl als Bytes (8 Bit), Wörter (16 Bit) und Langwörter (32 Bit) angesprochen werden. Mit Datenregistern können Sie alle Arten von logischen und arithmetischen Operationen ausführen. Die Adreßregister dagegen können nur als Wörter und Langwörter angesprochen, aber dafür auch als Zeiger auf Speicherstellen benutzt werden.

Wir führen nun nochmals die Addition unseres Beispielprogramms aus, aber berechnen das Ergebnis diesmal nicht im Speicher, sondern in einem Datenregister.

```
START   MOVE.L  #0,D0       ; D0 löschen (32 bit)
        ADD.L   #$1234,D0     ; $1234 zu D0
                              ; addieren
        RTS
```

Assemblieren und starten Sie dieses kleine Beispiel genauso wie vorhin, und schauen Sie sich das Register »D0« an, nachdem die Routine beendet wurde:

```
D0:00001234 00000000 00000000 00000000 00000000 00000000 00000000
A0:00000000 00000000 00000000 00000000 00000000 00000000 00012345
```

Die Nummer direkt nach »D0:« ist der Inhalt des Registers »D0«, der Wert daneben der des Registers »D1« usw. Schauen Sie auf »D0« - stimmt der Inhalt? Alle anderen Register können auch andere Werte als 0 enthalten. Das ist normal, schließlich werden die Register auch von anderen Programmen benutzt und ab und zu unaufgeräumt zurückgelassen. Deshalb ist es auch bei Registern wichtig, sie vor weiterer Benutzung mit dem Befehl »MOVE« zu laden.

Wir versuchen nun ein etwas fortgeschritteneres Beispiel, in dem auch die Adreßregister zusammen mit einigen neuen Befehlen benutzt werden.

Die neuen Befehle sind:

LEA	**Lade effektive Adresse**
TST	**Teste auf null**
BEQ	**Verzweige, wenn Ergebnis null ist**
BRA	**Verzweige immer (wie »GOTO« in Basic)**

Und hier ist das Programm - was tut es ?

```
START   LEA.L   BUFFER,A0  ; Lade Puffer-
                           ; adresse in A0

LOOP    TST.B    (A0)      ; Teste, ob A0 auf
                           ; ein Nullbyte zeigt
        BEQ     END        ; wenn 0, dann Ende
        ADD.B   #1, (A0)   ; addiere 1 zur
                           ; Adresse, auf die A0
                           ; zeigt
        ADD.L   #1,A0      ; addiere 1 zum
                           ; Zeiger A0
        BRA     LOOP       ; Verzweige zum
                           ; Schleifenbeginn

END     RTS                ; Rücksprung

BUFFER  DC.B '  G KKN',0   ; Der Puffer mit dem
                           ; Geheimtext
```

Assemblieren Sie den Quelltext, und starten Sie das Programm mit »J«. Geben Sie nun »H BUFFER« ein, und schauen Sie auf die Zeichen rechts. Wenn das Programm korrekt abgelaufen ist, steht dort eine kleine Nachricht für Sie.

In diesem Programm kam ein weiterer neuer Befehl hinzu: »DC.B«. Dies ist kein richtiger Maschinensprache-Befehl, sondern er teilt dem Assembler mit, den konstanten Wert, der dem Befehl folgt, in den Speicher zu schreiben. So könnten Sie beispielsweise schreiben:

```
DC.W 1000
```

Diese Anweisung schreibt die Konstante 1000 in den Speicher, und wie Sie in unserem Beispiel sehen konnten, ist es so auch möglich, Text in den Speicher zu schreiben. Jeder Buchstabe wird hierbei in eine Nummer umgewandelt, die auch **ASCII-Codes** genannt werden. »ASCII« steht für »American Standard Comitee for Information Interchange«, und »ASCII-Code« bezeichnet die von diesem Institut genormte Zuordnung des Zeichensatzes zu den 8-Bit-Werten.

Den gesamten ASCII-Code hier aufzulisten, lohnt nicht, da Sie eine Zusammenstellung des ASCII-Codes beispielsweise im Anhang des Amiga-Basic-Handbuchs finden. Einige Beispiele sollen an dieser Stelle zum besseren Verständnis dennoch gegeben werden:

0 = $30	A = $41	a = $61	. = $2e
1 = $31	B = $42	b = $62	; = $3b
2 = $32	C = $43	c = $63	? = $3f
3 = $33	D = $44	d = $64	

Wenn Sie die zugeordneten ASCII-Codes anderer Buchstaben wissen wollen, kann Ihnen auch hier der eingebaute Taschenrechner in ASM-ONE helfen.

Geben Sie zum Beispiel

```
>?'A'
```

ein, führt das zu folgender Ausgabe:

Hex	Dezimal	ASCII	Binär
$00000041	65	"...A"	%00000000.00000000.00000000.01000001

Nun wissen Sie bereits über einige Grundlagen der Programmierung in Maschinensprache Bescheid, und wenn Sie mehr über den Prozessor wissen wollen, sollten Sie das Kapitel »Der Motorola 68000« lesen.

5. Der Editor von ASM-ONE

Der eingebaute Editor ist mit einer Textausgabe von über 30000 Zeichen pro Sekunde recht schnell. Er unterstützt zudem viele nützliche Block- und Cursor-Steuerungsbefehle.

Um in den Editor zu wechseln, wählen Sie »**Edit**« im Assembler-Menü, drücken »**Amiga-Shift-E**« oder einfach »**Esc**«. Um das Editor-Fenster in der halben Größe zu öffnen, können Sie »**Ctrl+Esc**« statt »**Esc**« drücken.

Vom Editor aus können Sie alle Funktionen über das Menü »**Edit**« oder »**Shortcuts**« (Tastaturkürzel: »Amiga«, »Ctrl«, »Shift« oder »Alt« plus eine Taste) erreichen. Statt der »**Amiga**«**-Taste** können Sie wahlweise auch die »**Ctrl**«**-Taste** benutzen, so daß die meisten Kommandos mit einer Hand eingegeben werden können.

Ein **großer Buchstabe** im Menü bedeutet, daß Sie **zusätzlich** die »**Shift**«**-Taste** drücken müssen, um diese Funktion anzusprechen.

5.1 Zusammenfassung der Editor-Funktionen

Blockfunktionen:

»Amiga« oder »Ctrl« plus

b	- Block markieren
c	- Block kopieren
x	- Block ausschneiden
i , f	- Block einfügen
u	- Blockmarkierung löschen
l	- Block in Kleinschreibung wandeln
L	- Block in Großschreibung wandeln
y	- Block rotieren
k	- Benutzte Register anzeigen
w	- Block schreiben (zum Drucken »PRT:« angeben)

Suchen und Ersetzen:

»Amiga« oder »Ctrl« plus

S	- Suchen
s	- Suchen fortsetzen
R	- Suchen und Ersetzen
r	- Suchen und Ersetzen fortsetzen

Sprungbefehle:

»Amiga« oder »Ctrl« plus

!	- Markierung 1
@	- Markierung 2
#	- Markierung 3
1	- Springe zu 1
2	- Springe zu 2
3	- Springe zu 3
J	- Springe zur Markierung »;;«
j	- Springe zu Zeile ..

Cursor-Bewegung:

»Shift« plus

Hoch	- Seite hoch	
Runter	- Seite runter	
Links	- Zeilenbeginn	BOLN
Rechts	- Zeilenende	EOLN

»Amiga« oder »Ctrl« plus

a	- 100 Zeilen hoch
z	- 100 Zeilen runter
t	- Textbeginn
T	- Textende

»Alt« plus

links	- Wort links	LWORD
rechts	- Wort rechts	RWORD

Schnelles Löschen:

»Ctrl« plus

Del - Lösche bis Zeilenende

Back - Lösche ab Zeilenanfang

»Amiga« oder »Ctrl« plus

d - Zeile löschen (restaurierbar)

Sonstige:

»Amiga« oder »Ctrl« plus

m - Makro ausführen

M - Beginne Makrodefinition

g - Wort in Puffer übertragen
(das Wort kann mit »Cursor-Up« in der Eingabezeile in den Puffer geholt werden)

DEL - ein Zeichen löschen

BACKSPC - Zeichen vor dem Cursor löschen

Wenn »NumLock« selektiert ist, können Sie auch die numerischen Tasten benutzen, um den Cursor zu bewegen.

Zusätzliche Informationen zu den Editor-Befehlen:

J - Springe zur Markierung »;;«

Wenn Sie »;;«-Markierungen an einer oder mehreren Stellen in Ihrem Programmtext haben, **springt** »Amiga+Shift+j« zur nächsten folgenden »;;«-**Markierung**. Sie können dies anwenden, um die einzelnen Hauptteile Ihres Programms zu trennen.

!,@,# - Markierung 1,2,3

Mit diesen Zeichen können Sie **drei verschiedene Stellen** in Ihrem Programmtext **markieren**, ohne ihn ändern zu müssen, denn die Position wird nur intern gespeichert. Wenn Sie schnell zu einer bestimmten Stelle im Programmtext springen wollen, setzen Sie dort eines dieser Markierungszeichen, und springen Sie bei Bedarf direkt dorthin.

M - Beginne Makrodefinition

Die Definition eines Makros arbeitet wie ein Recorder, der alle Tastendrücke aufzeichnet. Wenn Sie also **mehrere Änderungen gleicher Art** vornehmen müssen, zeichnen Sie die dazu nötigen Tastenbedienungen ab dem Aufruf von »Amiga+Shift+m« auf und drücken in der Folge »Amiga+m«, um die anderen Änderungen ebenso vorzunehmen.

6. Die Befehlszeile

Die Befehls- oder Kommandozeile ist die Kommandozentrale von ASM-ONE. Sie erkennen die Befehlszeile am Prompt.

>

Folgende Befehle stehen Ihnen auf der Befehlszeile zur Verfügung:

Projekt:

ZS - Programmtext löschen
O - Programmtext restaurieren
R - Programmtext laden
RB - Binäre Daten laden
RO - Objektmodul laden
W - Programmtext speichern
WB - Binäre Daten speichern
WO - Objektmodul speichern
WL - Linkerfile speichern
I - Einfügen
U - Letzte Datei aktualisieren
ZF - Datei löschen
ZI - Include-Speicher löschen
WP - Preferences schreiben
=M - Zusätzlichen Arbeitsspeicher belegen
! - Assembler beenden

Editor:

T	[Zeile]	- Anfang des Textes (bzw. Zeile n)
B		- Ende des Textes
L	[text]	- Text suchen
ZL	[Zeilen]	- Zeilen ab Cursor-Position löschen
P	[Zeilen]	- Zeilen ab Cursor-Position ausgeben

Speicher:

M	[.size]	[addr]	- Speicher editieren
D	[addr]		- Disassemblieren
H	[.size]	[addr]	- Speicher hexadezimal anzeigen
N	[addr]		- Speicher als ASCII-Text anzeigen
@D	[addr]		- Disassembliere zeilenweise
@A	[addr]		- Assembliere zeilenweise
@H	[.size]	[addr]	- Hex-Dump zeilenweise
@N	faddr]		- ASCII-Dump zeilenweise
S	[.size]		- Suchen
F	[.size]		- Füllen
C	[.size]		- Kopieren
Q			- Vergleichen

Einfügen:

ID		- Disassemblierung einfügen
IH	[.size]	- Hexadezimal einfügen
IN		- ASCII einfügen

Assembler:

A	- Programmtext im Editor assemblieren
@A [addr]	- Assemblieren im Speicher
AO	- Optimierend assemblieren
AD	- Mit Debug-Informationen assemblieren
=S	- Symboltabelle erzeugen

Monitor:

J [addr]	- Subroutine anspringen (JSR)
G [addr]	- Programm direkt starten (JMP)
K [steps]	- Einzelschrittmodus
X [register]	- Register anzeigen bzw. editieren
ZB	- Breakpoints löschen

Diskette:

RS [drive]	- Sektor lesen
RT [drive]	- Spur lesen
WS [drive]	- Sektor schreiben
WT [drive]	- Spur schreiben
CC [drive]	- Bootblock-Checksumme berechnen
E	- Externe Dateien laden
V [path]	- Verzeichnis anzeigen

Sonstige:

>	- Ausgabe umleiten (z. B. zu »PRT:« oder »DFx:«)
? [expr]	- Numerischen Ausdruck berechnen

Auch hier bedürfen einige der aufgeführten Befehle noch einer genaueren Erläuterung:

Projekt:

ZS - Programmtext löschen / Zap Source

Löscht den Programmtext, den Kopierpuffer und den Code. Der Programmtext kann mit dem Befehl »O« (»Old source«) wiederhergestellt werden, jedoch nur, wenn Sie den Editor danach noch nicht wieder aufgerufen haben.

O - Programmtext restaurieren / Old source

Dieser Befehl ermöglicht es Ihnen, das Kommando »ZS« rückgängig zu machen. Er kann auch benutzt werden, um den Programmtext nach einem Programmabsturz noch zu retten. Ein guter Tip: Dazu legen Sie Ihren Arbeitsspeicher jedesmal an dieselbe absolute Adresse.

R - Programmtext laden / Read

Liest eine Datei in den Editor ein, die jede Art von Text enthalten kann. Normalerweise wird die Extension ».s« (wie »Source«) an den Namen der Datei angehängt, wenn Sie dies nicht wünschen, können Sie die Extension manuell löschen oder das Flag »Source.S« im Menü »Preferences« zurücksetzen.

Der Befehl »R« (»Read«) löscht einen im Editor befindlichen Text. Benutzen Sie »I« (»Insert«), um Programmteile in bereits im Editor befindliche Textteile einzufügen.

RB - Binäre Daten laden / Read Binary

Mit diesem Befehl können Sie binäre Daten an eine bestimmte Speicheradresse laden. Nachdem Sie den Dateinamen eingegeben haben, werden Sie wie folgt nach der Start- und Endadresse gefragt:

```
BEG>
END>
```

»BEG« (Beginn) ist die erste Speicheradresse, die von der Datei gefüllt werden soll, »END« (Ende) die letzte.

```
BEG>$70000
END>$71000
```

beispielsweise liest die ersten $1000 = 4096 Byte des binären Files in den Speicher, beginnend mit der Adresse »$70000«. Wenn Sie die gesamte Datei laden wollen, dann ignorieren Sie die Frage »END« einfach, indem Sie »Return« drücken:

```
BEG>$70000
END>
```

Die binäre Datei wird so in ihrer ganzen Länge geladen.

RO - Objektmodul laden / Read Object

Liest eine ausführbare Datei (»Executable«, also Programme wie »DIR«, »LIST«, ASM-ONE etc.). Für die Datei wird neuer Speicher belegt; sie wird an die Speicheradresse geladen und für diese Adresse reloziert, die anschließend ausgegeben wird. In den meisten Fällen können Sie das geladene Programm an dieser Adresse starten.

Programme mit Parametern wie dem CLI-Befehl »DIR« und ähnlichen erwarten diese Parameter vom aufrufenden

CLI, so daß diese Übergabeparameter anhand der Register »A0« (Zeiger auf Parameter) und »D0« (Länge der Parameter) simuliert werden müssen.

Da ein auf diese Art nachgeladenes Programm Speicher belegt, wird der vom letzten geladenen Programm belegte Speicher freigegeben, wenn Sie ein neues Objekt-File nachladen. Sie können den Speicher ganz freigeben, indem Sie ein Objekt ohne Namen ("") laden.

W - Programmtext speichern / Write

Dieser Befehl speichert den im Editor befindlichen Programmtext als ASCII-Datei. Nahezu alle anderen Editoren und Textverarbeitungen können ASCII-Texte laden, so daß Sie durchaus auch andere Texte als Programmtexte mit dem Editor von ASM-ONE schreiben können.

Normalerweise wird die Extension ».s« (wie »Source«) an den Dateinamen angehängt. Wenn Sie dies nicht wünschen, können Sie die Extension manuell löschen oder das Flag »Source.S« im Menü »Preferences« zurücksetzen. Bei zurückgesetztem Flag können Sie mit dem Befehl »!« und nachfolgender Extension in der Preference-Datei eine Extension Ihrer Wahl festlegen.

WB - Binäre Daten speichern / Write Binary

Schreibt binäre Daten von der angegebenen Speicheradresse als Binär-File auf Diskette. Nach der Eingabe des Dateinamens fragt das Programm nach Start- und Endadresse:

```
BEG>
END>
```

»BEG« ist der Start des zu speichernden Bereichs, »END« bezeichnet die erste Adresse nach dem Bereich.

```
BEG>$70000
END>$71000
```

schreibt beispielsweise $1000 = 4096 Byte aus dem Speicher von »$70000« an in das Binär-File.

WO - Objektmodul speichern / Write Object

Wenn ein Programmtext assembliert wurde, ist es mit »WO« möglich, den erzeugten Code als ein ausführbares Programm auf Diskette zu speichern (auch Loadfile oder Executable genannt). Wenn Sie ein Linker-File speichern wollen, können Sie »WL« (»Write Linkfile«) benutzen.

I - Einfügen / Insert

Dieser Befehl ermöglicht es Ihnen, ein neues Stück Programmtext in einen im Editor befindlichen Text einzufügen. Der Name des im Editor befindlichen Programmtextes wird dadurch nicht geändert. Sehen Sie für detaillierte Informationen unter »R - Programmtext laden/Read« nach.

U - Letzte Datei aktualisieren / Update file

Schreibt die im Editor befindliche Datei unter demselben Namen, unter dem es geladen wurde, auf Diskette; wie »W« (»Write«).

ZF - Datei löschen / Zap file

Löscht eine Datei von der Diskette.

ZI - Include-Speicher löschen / Zap include memory

Include-Files werden beim Assemblieren in den Speicher gelesen (siehe auch unter »Assembler-Direktiven«). Um wiederholte Assemblierungen zu beschleunigen, werden sie danach im Speicher gehalten, um nicht jedesmal neu nachgeladen werden zu müssen. Um diesen Speicher zwischenzeitlich freizugeben, können Sie »ZI« benutzen.

WP - Preferences schreiben / Write preferences

Diese Funktion erzeugt die Datei »ASM-One.Pref«, die den Status der Flags im Menü »Preferences« enthält. Wenn Sie diese Datei in den Editor laden, kann sie beispielsweise so aussehen:

```
-RS-L7+NL+AA+RL+WB
```

Sie können die »Preferences«-Datei ändern, so daß sie mehr bewirkt, als nur die Flags zu setzen. Zum Beispiel wie folgt:

```
-RS-L7+NL+AA+RL+WB\F\200\
```

Diese Einstellung belegt beim Start automatisch 200 KByte Fast-Memory als Arbeitsspeicher. Das Zeichen »\« entspricht dabei einem »Return«.

=M - Zusätzlichen Arbeitsspeicher belegen

Nehmen wir einmal an, daß Sie 200 KByte Speicher als Arbeitsspeicher belegt haben, was aber nicht ausreicht. Mit dieser Funktion können Sie den Arbeitsspeicher nachträglich erweitern, sofern noch ausreichend freier Speicher, angrenzend an den bereits reservierten, verfügbar ist. Ist dies jedoch nicht möglich, müssen Sie ASM-ONE neu starten, um neuen Speicher zu belegen. Siehe hierzu auch »! - Assembler beenden/Quit assembler«.

! - Assembler beenden / Quit assembler

Wenn Sie diesen Befehl eingeben, stehen Ihnen zwei Möglichkeiten offen: ASM-ONE zu verlassen oder neu zu starten. In beiden Fällen wird der belegte Speicher wieder freigegeben, und Ihr Text im Editor ist verloren. Bei einem Neustart verhält sich ASM-ONE, als ob es neu geladen wurde.

Editor:

T - Anfang des Textes (bzw. Zeile n)

Springt zur Zeilennummer, die nach »T« angegeben ist, ansonsten zum Textbeginn. Beispiele:

```
T100     ; Springt zu Zeile 100

T        ; Springt zum Anfang des Textes

T-1      ; Springt zur letzten Zeile
```

L - Text suchen / Look for text

Sucht im Quellcode nach dem Text, der nach »L« angegeben ist. Beispiel:

```
LMOVE
```

sucht nach allen Move-Befehlen im Programmtext.

Um nochmals nach demselben Begriff zu suchen, genügt die Eingabe von:

```
L
```

ZL - Zeilen ab Cursor-Position löschen / Zap Lines

Löscht die angegebene Zahl von Zeilen ab der aktuellen Cursor-Position. Beispiel:

```
ZL100
```

löscht 100 Zeilen von der Cursor-Position an.

```
ZL-1
```

löscht alle Zeilen von der Cursor-Position an.

P - Zeilen ab Cursor-Position ausgeben / Print lines

Gibt von der Cursor-Position an die Anzahl von Zeilen aus, die nach »P« angegeben wurde. Um diese Zeilen auf den Drucker auszugeben, wählen Sie »PrinterDump« im Submenü »Preferences« oder drücken »Ctrl-P«. Informationen zur Installation des Druckertreibers finden Sie in Anhang A. Beispiel:

```
P100
```

gibt 100 Zeilen von der Cursor-Position an aus.

```
P-1
```

gibt alle Zeilen nach der Cursor-Position aus.

Memory:

M - Speicher editieren / Memory edit

Fügt Text oder Hexadezimalwerte an einer bestimmten Speicherstelle ein.

D - Disassemblieren / DisAssemble

Wechselt in den Disassemblier-Modus, der Ihnen einen Einblick in den Code gibt. Sie können, genauso wie bei Texten im Textmodus, hoch- und herunterscrollen, springen und einzelne Befehle ändern. Wenn Sie eine solche Änderung rückgängig machen wollen, reicht ein Druck auf »Esc« (siehe auch »Monitorfunktionen«).

H - Speicher hexadezimal anzeigen / HexDump

Wechselt in den Hexdump-Modus des Monitors. Auch hier bekommen Sie einen Einblick in die Speicherbelegung (hexadezimal und im ASCII-Code). Sie können nach Belieben scrollen, springen und editieren.

Wenn Sie langwortweise Werte ändern wollen, ist der Befehl »M.L« interessanter, da Sie damit das ganze Langwort auf einmal, anstatt nur nibbleweise, ändern können (siehe auch hier »Monitorfunktionen«).

N - Speicher als ASCII-Text anzeigen / AsciiDump

Wechselt in den ASCII-Modus des Monitors. Sie können frei im Speicher editieren, der in Reihen von je 64 ($40) Buchstaben angezeigt wird, indem Sie den Cursor auf die richtige Stelle bewegen und den Text normal eingeben.

@D - Disassembliere zeilenweise / Disassemble Mem

Disassembliert zwölf Zeilen Programmcode ab der angegebenen Adresse. Wenn keine Adresse angegeben ist, wird mit der Disassemblierung bei der letzten Adresse fortgefahren. Benutzen Sie die Funktion »Disassemblierung einfügen«, um die so disassemblierten Code-Abschnitte in Ihr Programm einzufügen.

@A - Assembliere zeilenweise / Assemble Mem

Assembliert Befehle direkt in den Speicher, wie in den alten C64-Monitoren.

@N - ASCII-Dump zeilenweise / Ascii dump

Zum Einfügen eines ASCII-Dumps in Ihren Programmtext siehe auch bei »Insert ASCII Dump«.

@H - Hex-Dump zeilenweise / Hex dump

Zum Einfügen eines Hex-Dumps in Ihren Programmtext siehe auch bei »Insert Hex Dump«.

S - Suchen / Search in memory:

Die Eingabe geht wie folgt vor sich:

```
>S
BEG>$10000
END>$20000
```

Nachdem Sie die Start- und Endadressen eingegeben haben, können Sie die Daten eingeben, die Sie suchen wollen, also beispielsweise

```
DATA>123 4321.L "HELLO" $5432.W %100101.B
```

Default beim Suchen ist die Byte-Größe.

F - Füllen / Fill memory

Seien Sie mit diesem Befehl vorsichtig: Wenn Sie den falschen Speicherbereich füllen, kann das Ihren Amiga zum Absturz bringen.

C - Kopieren / Copy memory

Kopiert einen Speicherbereich an eine andere Adresse. Auch hier sollten Sie wie beim Befehl »Fill« vorsichtig sein.

Q - Vergleichen / Compare memory

Vergleicht zwei Speicherbereiche. Wenn die Inhalte nicht gleich sind, wird die Adresse des ersten ungleichen Bytes ausgegeben.

Einfügen:

ID - Disassemblierung einfügen / Insert DisAssemble

Dies ist ein sehr mächtiger Befehl, der es Ihnen ermöglicht, den Speicher zu disassemblieren und diesen Bereich direkt in Ihren Programmtext einzufügen. Der neue Code wird so weit wie möglich mit Labels versehen, damit Sie ihn wie jeden anderen Programmtext direkt verwerten können.

IH - Hexadezimal einfügen / Insert Hexdump

Erlaubt Ihnen, einen Speicherbereich in Form von DC-Pseudo-Opcodes in Ihren Programmtext einzubinden.

IN - ASCII einfügen / Insert Ascii

Erlaubt Ihnen, einen Textbereich als DC.B-String-Tabelle in Ihren Programmtext einzufügen. Wenn Nicht-ASCII-Werte auftauchen, wird an dieser Stelle das entsprechende Hex-Byte verwendet.

Assembler:

A - Programmtext im Editor assemblieren

Der normale Assembler-Start, wie auf »Amiga-Shift-A«.

@A - Assemblieren im Speicher / Assemble to memory

Siehe hierzu die »Memory«-Sektion.

AO - Optimierend assemblieren / Assemble optimize

Assembliert Ihren Programmtext normal, jedoch wird versucht, alle Branches auf die kürzere ».S«-Variante (Distanzangabe als Byte) zu optimieren. Wenn der Branch weiter als -128 bis +127 Bytes geht, wird er in einen Branch mit Wortdistanz umgewandelt.

AD - Mit Debug-Informationen assemblieren / Assemble debug

Assembliert den Programmtext unter Erzeugung der Debug-Informationen. Bei aktiviertem Debugger wird automatisch auf diese Weise assembliert.

=S - Symboltabelle erzeugen / Symbol table print

Zeigt Ihnen nach der Assemblierung alle globalen Labels an. Diese können auch auf dem Drucker ausgegeben werden, indem Sie die Option »PrinterDump« aktivieren. Die Symboltabelle wird automatisch erzeugt, wenn Sie die Option »Listfile« wählen.

Monitor:

J - Subroutine anspringen / Jump to address

Springt die spezifizierte Adresse als Subroutine an (per »JSR«). Wenn keine Adresse angegeben ist, wird an den Start des Programms gesprungen. Um zum Beispiel an das Label »START« zu springen, reicht die Anweisung

```
>JSTART
```

G - Programm direkt starten / Go to address

Entspricht dem Befehl »J« allerdings kann den Programmlauf nur noch ein Breakpoint oder ein illegales Kommando stoppen. (»G« springt per »JMP« statt »JSR« an die angegebene Adresse.)

K - Einzelschrittmodus / Single step n steps

Führt n Programmschritte vom aktuellen Programmzähler an im Einzelschrittmodus aus. In der Beschreibung des Debuggers wird diesem Thema mehr Raum gewidmet.

X - Register anzeigen bzw. editieren

Kann in zwei verschiedenen Varianten angewendet werden:

```
>X    ; Alle Register anzeigen

>XD2  ; ein Register editieren (Hier: D2)
```

Wenn Sie »X« alleine eingeben, werden Ihnen alle Register angezeigt, inklusive »USP«, »SSP«, »SR« und »PC«. Die Statusregister-Flags werden auch als Buchstaben angezeigt. Alle Änderungen seit der letzten Registeranzeige werden invers angezeigt (in der Druckerausgabe unterstrichen).

Diskette:

RS - Sektor lesen / Read sector

Liest einen Sektor in den Speicher. Eine Amiga- Diskette hat je 80 Spuren auf beiden Seiten, die jeweils in elf Sektoren unterteilt sind. Das macht 80*2*11=1760 Sektoren zu jeweils 512 Byte. Der Bootblock besteht aus zwei Sektoren, beginnend mit Sektor 0. Der Diskettenname befindet sich im Root-Block der Diskette, der sich normalerweise in Sektor 880 befindet. Wenn Sie nun den zwei Sektoren langen Bootblock vom Laufwerk »DF1:« an Adresse »$70000« in den Speicher einlesen wollen, geben Sie ein:

```
>RS1              ; Laufwerk 1
RAM PTR>$70000    ; Zeiger auf RAM-Adresse
DISK PTR>0        ; Nummer des ersten Sektors
LENGTH>2          ; Anzahl der Sektoren
```

Eine Einschränkung ist zu beachten: Daten aus direktem Diskettenzugriff können nur ins Chip-Memory geladen werden.

RT - Spur lesen / Read track

Siehe »RS«.

WS - Sektor schreiben / Write track

Siehe »RS«.

WT - Spur schreiben / Write track

Siehe »RS«.

CC - Bootblock-Checksumme berechnen

Wie unter »RS« bereits erwähnt, ist der Boot-Block 1024 Byte lang und wird vom System zuerst geladen. Die Informationen des Boot-Blocks teilen dem System mit, von welcher Art die Diskette ist. (Kickstart, DOS\0 (Old File System) oder DOS\1 (Fast File System)).

Der Boot-Block kann auch ein Programm enthalten. Um festzustellen, ob es ausführbar ist, überprüft das System eine Checksumme. Mit »CC« können Sie diese Checksumme (zum Beispiel nach Änderungen) neu berechnen. Dies ist nützlich, wenn Sie einen eigenen Boot-Block auf die Diskette installiert haben.

Um beispielsweise die Checksumme für den Bootblock auf »DF1:« zu berechnen, geben Sie ein:

```
>CC1
```

E - Externe Dateien laden / Extern files load:

Dateien, die im Programmtext mit einer »Extern«-Direktive markiert wurden, werden nachgeladen, wenn Sie »E« eingeben. Wenn Sie nur einige bestimmte externe Dateien nachladen wollen, können Sie hinter dem Befehl noch eine Nummer angeben. Nähere Informationen finden Sie unter »Extern-Direktive«.

V - Verzeichnis anzeigen / View directory

Zeigt das Verzeichnis von dem nach »V« angegebenen Pfad aus an. Dieser Befehl ist nötig, weil Sie sonst bei abgeschalteter »Req-Library« keine andere Möglichkeit hätten, um herauszufinden, was sich auf der Diskette befindet. Wenn Sie beispielsweise den Inhalt der Diskette in »DF0:« wissen möchten, geben Sie

```
VDF0:
```

ein. Wenn Sie das Verzeichnis nicht angezeigt haben möchten, sondern nur das Hauptverzeichnis wechseln wollen, geben Sie ein Leerzeichen zwischen Befehl und Pfad an.

```
V DF0:
```

Wenn Sie sich das letzte angezeigte Verzeichnis nochmals in sortierter Form ansehen möchten, reicht ein einfaches »V«.

Sonstige:

> - Ausgabe umleiten / Specify Output:

Sie können diesen Befehl benutzen, um die Ausgabe des Programms auf einen Drucker oder in eine Datei umzuleiten. Wenn Sie als Datei ausgeben, erhält diese die Extension ».TXT«. Eine so erzeugte Datei ist ein Standard-Textfile und kann in den Editor geladen werden. Um eine solche Datei- oder Druckerausgabe zu beenden, müssen Sie den Befehl lediglich erneut (nun mit einem leeren Ausgabepfad) eingeben.

Wenn Sie den auf den Bildschirm ausgegebenen Text in eine Datei namens »DIARY« speichern möchten, schreiben Sie

```
>>
FILENAME>DIARY
```

Um die Ausgabe zu beenden, geben Sie beim nochmaligen Befehl lediglich »Return« bei der Abfrage des Dateinamens ein.

? - Numerischen Ausdruck berechnen / Calculate value

Sie können bei dieser Funktion alle üblichen Operatoren und alle definierten Labels verwenden. Das Resultat wird in hexadezimaler, dezimaler, binärer und in ASCII-Darstellung ausgegeben.

Um die Summe von 123 und 321 zu berechnen, schreiben Sie

```
>?123+321
```

Um das Quadrat von »D0« berechnen , schreiben Sie

```
>?D0*D0
```

7. Der Assembler

Der 'Assembler von ASM-ONE ist einer der schnellsten auf dem Markt. ASM-ONE assembliert mit einer Geschwindigkeit von 50000 bis 70000 Zeilen pro Minute auf einem Standard-Amiga. Es ist kompatibel zum Metacomco-Makro-Assembler, wodurch Sie auch alte Programmcodes ohne Probleme mit ASM-ONE übersetzen können.

Haben Sie einen Programmtext in den Editor eingelesen, starten Sie den Assembliervorgang durch »Assemble« im Menü »Assembler«, durch »Amiga-Shift-A« oder folgende Eingabe in der Kommandozeile:

>a

Wenn beim Assemblieren ein Fehler auftritt, werden Ihnen die Zeilennummer und der Inhalt der fehlerhaften Zeile sowie eine Fehlermeldung ausgegeben.

Sie können die Assemblierung mit »Ctrl-C« vorzeitig beenden.

Mit folgenden Optionen, die als Flags gesetzt werden, läßt sich der Vorgang der Assemblierung beeinflussen:

1 - Listfile erzeugen

2 - Seitenweise anzeigen

3 - Seitenweise stoppen

4 - Alle Fehler anzeigen

5 - Debug

6 - Label:

7 - UCase = LCase

Im einzelnen bedeuten die Flags folgendes:

(1) Wenn Sie dieses Flag setzen, wird ein Listfile erzeugt. Die Flags (2) und (3) beeinflussen die Ausgabe dieses Listfiles.

(2) und **(3)** Siehe Flag (1).

(4) Wenn dieses Flag gesetzt ist, assembliert ASM-ONE den gesamten Programmtext, ohne beim ersten Fehler zu stoppen. Der Assembler gibt Fehlerzeilen und Fehlermeldungen wie gewohnt aus, fährt jedoch mit der Assemblierung fort. Sie können diese Fehlerliste mit der Option »PrinterDump« zu Papier bringen (»Ctrl-P«).

(5) Dieses Flag aktiviert die Option »Debug«. Hierdurch wird zwar mehr Speicher benötigt, aber wenn Sie den Debugger häufig benutzen, können Sie ohne neue Assemblierung in den Debugger wechseln.

(6) Ist dieses Flag gesetzt, müssen alle Labels mit einem Doppelpunkt »:« enden. Der Vorteil dieses Verfahrens ist, daß die Assembler-Befehle direkt in der ersten Spalte beginnen können.

Da einige Assembler dieses Verfahren verwenden, können Sie einen Quellcode von einem solchen Assembler auch mit ASM-ONE übersetzen.

(7) Dieses Flag ist normalerweise gesetzt. Sie können es deaktivieren, wenn der Assembler strikt zwischen Groß- und Kleinschreibung unterscheiden soll.

Ist ein Programm fertig assembliert, können Sie es als »Executable« speichern, indem Sie die Funktion »WO« (»Write Object«) nutzen. Wenn Sie externe Referenzen in Ihrem Programm angegeben haben, müssen Sie das Programm mit »WL« als Linkfile speichern und anschließend noch linken.

7.1 Programme

Jede Zeile Ihres Quelltextes muß einem der folgenden Typen entsprechen:

1 - Leere Zeile oder Kommentar

2 - Befehl (Opcode)

3 - Assembler-Steuerungsdirektive (Pseudo-Opcode)

Einige Beispiele für Kommentare:

(A) Eine Zeile, die mit einem **Sternchen** beginnt, ist eine Kommentarzeile. Zum Beispiel:

```
* Dies ist mein Programm
```

(B) Ein **Semikolon** »;« an beliebiger Stelle deklariert den Rest der Zeile als Kommentar. Zum Beispiel:

```
subq.l   #1,d0      ; Rest = Rest - 1
cmp.l    d1,d0      ; gleich ?
```

(C) Ein **Text-String**, der einem kompletten Befehl oder einer Direktive folgt, ist ein Kommentar. Zum Beispiel:

```
moveq    #0,d0      ; Zähler löschen
```

Im allgemeinen folgt eine Zeile dieser Struktur:

```
[<Label>] <Opcode> [<Operand>[,<Operand>]...] [<Kommentar>]
```

Wie Sie sehen, läßt sich eine Zeile in Felder unterteilen.

Label	Opcode	Operanden	Kommentar

Die Felder müssen mit einem oder mehreren Leerzeichen oder Tabulatoren getrennt sein.

7.1.1 Das Label-Feld

Ein Label ist ein vom Benutzer definiertes Symbol und beginnt entweder

1. **in der ersten Spalte**
2. **oder in einer anderen Spalte, wenn es auf »:« endet**

Labels wird die Adresse des ersten Bytes der folgenden Instruktion zugewiesen. Zwei Labels darf nicht derselbe Name zugewiesen werden, andernfalls wird ASM-ONE die Fehlermeldung »doppeltes Symbol« ausgeben:

```
** Double Symbol
```

Lokale Labels:

Ein lokales Label entspricht einem normalen Label, ist dem Assembler aber nur in einem Teil des Programms bekannt, so daß Sie denselben Namen mehrmals verwenden können. Ein lokales Label wird ebenso wie ein globales Label definiert, jedoch mit einem Punkt als erstes Zeichen, wie zum Beispiel

```
.label1
```

Ein lokales Label darf nur zwischen dem ihm vorangehenden und dem ihm folgenden globalen Label angesprochen werden. Ein Beispiel dafür:

Label	Opcode	Operanden	Kommentar
clear	moveq	#0,d0	
	moveq	#100-1,d0	
	lea	memory,a0	; Zu löschender ; Speicher
.loop	move.l	d0,(a0)+	
	dbf	d1,.loop	
end	rts		

In diesem Beispiel ist das lokale Label nur zwischen »clear« und »end« bekannt, so daß Sie ».loop« so auch an anderen Stellen in Ihrem Quelltext verwenden können.

7.1.2 Das Befehlsfeld

Ein Befehl darf nicht in der ersten Spalte beginnen, es sei denn, daß Sie das entsprechende Label-Flag (6) bei der Assemblierung setzen. Ihm muß mindestens ein Leerzeichen oder ein Tabulator oder ein von einem solchen gefolgtes Label vorangehen.

Der Befehl ist entweder ein vor- oder benutzerdefiniertes Symbol. Möglich sind:

1) ein Motorola-MC68000-Assembler-Befehl

2) eine Assembler-Direktive

3) ein Makro-Befehl

Dem Befehl darf eine Größenangabe folgen. Eine Größenangabe kann eine der folgenden sein:

.B - Byte-Operand
.W - Wort-Operand
.L - Langwort-Operand

Bei Branches:

.W - Branch mit 16-Bit-Offset (auch ».L« für »Long Branch«)
.B - Branch mit 8-Bit-Offset (auch ».S« für »Short Branch«)

Anmerkung des Übersetzers: Die Alternativen ».L« und ».S« sind nur aus Kompatibilitätsgründen zu älteren Assemblern implementiert. In Hinblick auf Aufwärtskompatibilität zu Assemblern für 68020/30 und höhere Prozessoren sollten Sie in Ihren Programmtexten auf ».L« verzichten, das auf 68020-Assemblern einen relativen Branch mit 32-Bit-Offset bezeichnet.

Bei der Größenangabe muß darauf geachtet werden, daß der zugehörige Befehl diese unterstützt. Adreßregister lassen sich beispielsweise nur wort- und langwortweise, nicht aber Byteweise ansprechen.

7.1.3 Das Operandenfeld

Dieses Feld kann je nach Befehl nur einen oder mehrere Operanden beinhalten. Jeder Operand muß dabei durch mindestens ein Leerzeichen oder einen Tabulator vom Befehl getrennt sein, und jeder Operand wird mit einem Komma vom vorhergehenden getrennt. Diesem Komma dürfen in einer Erweiterung des Motorola-Standards noch weitere Leerzeichen oder Tabulatoren folgen, da es oftmals angenehm ist, die Operanden in Spalten zu gruppieren.

Anmerkung des Übersetzers: Verwenden Sie dieses Feature nicht, wenn Ihr Quelltext auch auf anderen Assemblern als ASM-ONE assembliert werden soll.

7.1.4 Kommentare

Nach einem syntaktisch korrekten und mit mindestens einem Leerzeichen oder Tabulator beendeten Kommando wird alles weitere als Kommentar angesehen.

Anmerkung des Übersetzers: Die meisten anderen Assembler als ASM-ONE geben in diesem Fall einen Fehler aus. Sie sollten daher eventuell folgende Kommentare dennoch mit einem Semikolon beginnen.

7.1.5 Terme

Ein Term ist eine Kombination von algebraischen Operatoren, Operanden, Konstanten und Klammern. Er kann einen konstanten, relativen oder booleschen Wert annehmen. In letzterem Fall wird »Wahr« durch »-1«, »Falsch« durch »0« repräsentiert.

ASM-ONE kennt folgende Operatoren:

1. Vorzeichenminus, logisches Nicht	**-, ~**
2. Links-rechts-Verschiebung, Potenz	**<<, >>, ^**
3. Logisches Und, Oder, Exklusiv-Oder	**&, !, ~**
4. Multiplikation, Division	***, /**
5. Addition, Subtraktion	**+, -**
6. Vergleichsoperatoren	**>, >=, =, <=, <, <>**

Bei der Auswertung eines Terms werden die Operanden in der Reihenfolge ihrer Priorität behandelt. In der vorausgegangenen Liste haben die niedrigsten Nummern die höchste Priorität.

Um diese Reihenfolge zu ändern, können Sie Klammern benutzen, wobei sowohl eckige als auch runde Klammern erlaubt sind.

Anmerkung des Übersetzers: Die eckigen Klammern sind zwar aus Kompatibilitätsgründen zulässig, sollten aber nicht verwendet werden. Zum Beispiel

`[1+2]*3 = 9` oder `(1+2)*3 = 9`

aber

`1+2*3 = 7`

Operanden oder Symbole:

Ein Symbol ist ein String, der aus bis zu 100 Zeichen bestehen darf, wobei jedes einzelne einem der vier folgenden Typen zugehören muß. Einzige Einschränkung ist, daß der jeweils erste nur aus den Gruppen 1 bis 3 stammen darf.

1. **Buchstabe** von »A« bis »Z« und »a« bis »z«; strikte Unterscheidung zwischen Groß- und Kleinbuchstaben ist optional über ein Flag aktivierbar.
2. **Unterstrich** »_«
3. **Punkt** ».«
4. **Ziffer** von »0« bis »9«

Symbole, die mit einem Punkt beginnen, werden als lokale behandelt (siehe »Lokale Labels«). Ein Symbol kann in einer der folgenden Varianten definiert werden.

Absoluter Wert:

EQU - Das Symbol wird mit einem absoluten Wert gleichgesetzt.

SET - Das Symbol wird als absoluter Wert definiert.

RS - Definiert Offsets in Strukturen.

<Label> - Das Symbol wird als Label in einem absoluten Programm genutzt (ein Programm, das die Direktive »ORG« nutzt, ist absolut).

Relativer Wert:

EQU - Das Symbol wird mit einem relativen Wert gleichgesetzt.

SET - Das Symbol wird als relativer Wert definiert.

<Label> - Das Symbol wird als Label genutzt.

Registersymbole:

EQUR - Das Symbol wird mit einem Register gleichgesetzt.

REG - Das Symbol wird mit einer Registerliste gleichgesetzt.

Zahlen:

Auch Zahlen können wie jedes andere Konstantensymbol in Termen auftauchen. Es gibt fünf verschiedene Darstellungsweisen von numerischen Werten, zum Beispiel:

1. Dezimal	**1234, -234**
2. Hexadezimal	**$fab, -$fc**
3. Oktal	**@176**
4. Binär	**%1001001**
5. ASCII	**'abcd', "abcd", \`abcd\`**

ASCII-Strings mit weniger als vier Zeichen werden rechtsbündig interpretiert. Um Anführungszeichen in einem ASCII-String zu verwenden, sollten Sie sie wie folgt in eins der beiden anderen Anführungszeichen einbetten oder einfach doppelt schreiben:

Aus	'	wird	"'"	oder	''
aus	"	wird	'"'	oder	""
und aus	\`	wird	"\`"	oder	\`\`

7.2 Direktiven

Die Assembler-Direktiven (auch Pseudo-Opcodes genannt) sind keine Befehle, die Code erzeugen, sondern Anweisungen an den Assembler. Einzige Ausnahmen sind »DC« und »DCB«.

Die Direktiven im Überblick

Zur Assemblierungskontrolle:

SECTION	Beginn einer Programmsektion
RORG	Relativer Programmstart
ORG	Absoluter Programmstart
LOAD	Ladeadresse (absolut) setzen
OFFSET	Definiert Offsets
ENDOFF	Beendet Offset-Tabelle
END	Beendet Programm
BASEREG	Setzt Basisregister

Zur Datendefinition:

DC	Definiert Konstante
DCB	Definiert Konstantenblock
DS	Definiert Freiraum für Variablenspeicherung
BLK	Block (siehe »DCB«)
DR	Definiert relativen Wert

Symboldefinition:

EQU	Weist dauerhaften Wert zu
SET	Weist temporären Wert zu
EQUR	Weist Register zu
REG	Weist Registerliste zu
RS	Weist relativen Wert zu
RSRESET	Setzt Zähler für relative Offsets zurück
RSSET	Setzt Zähler für relative Offsets

Makro-Direktiven:

MACRO	Beginnt eine Makro-Definition
NARG	Spezielles Symbol (Zahl der Parameter)
ENDM	Beendet Makro-Definition
MEXIT	Verläßt Makro
CMEXIT	Verläßt Makro, wenn Rekursionstiefe erreicht
REPT	Beginnt Code-Wiederholung
ENDR	Beendet Code-Wiederholung

Bedingte Assemblierung:

CNOP	Bedingtes »NOP« zur Ausrichtung an Wort- oder Langwortgrenzen
EVEN	Erzwingt Fortsetzung auf gerader Adresse
ODD	Erzwingt ungerade Adresse
IFEQ	Assembliert, wenn Term = 0 (EQual)
IFNE	Assembliert, wenn Term <> 0 (Not Equal)
IFGT	Assembliert, wenn Term > 0 (Greater Than)
IFGE	Assembliert, wenn Term >= 0 (Greater or Even)
IFLT	Assembliert, wenn Term < 0 (Less Than)
IFLE	Assembliert, wenn Term <= 0 (Less or Even)
IF	Assembliert, wenn Term logisch wahr
IFC	Assembliert, wenn Strings identisch sind
IFNC	Assembliert, wenn Strings nicht identisch sind
IFD	Assembliert, wenn Symbol definiert ist
IFND	Assembliert, wenn Symbol nicht definiert ist.
IFB	Assembliert, wenn leer
IFNB	Assembliert, wenn nicht leer
IF1	Assembliert, wenn in »Pass1«
IF2	Assembliert, wenn in »Pass2«
ELSE	Alternativer Code, wenn Bedingung nicht erfüllt
ENDC	Beendet »IF«-Block

Ausgabekontrolle:

PAGE	Beginnt neue Seite im Listfile
NOPAGE	Schaltet Seitentrennung ab
LIST	Aktiviert Listing
NOLIST (NOL)	Beendet Listing
LLEN	Setzt Zeilenlänge
PLEN	Setzt Seitenlänge (Zeilenzahl)
SPC	Erzeugt n leere Zeilen
TTL	Setzt Programmtitel
FAIL	Generiert Assembler-Fehler
MASK2	(nur aus Kompatibilitätsgründen implementiert)
PRINTT	Gibt String auf dem Bildschirm aus
PRINTV	Ausgabe eines Wertes auf dem Bildschirm

Externe Symbole:

XDEF	Externe Definition
XREF	Externe Referenz
ENTRY	Siehe »XDEF«
EXTRN	Siehe »XDEF«
GLOBAL	Siehe »XDEF«

Allgemeine Direktiven:

JUMPPTR	Setzt Startpointer
INCBIN	Binär-File einfügen
IMAGE	Siehe »INCBIN«
INCLUDE	Sourcefile einfügen
INCDIR	Pfad für Includes eingeben
>EXTERN	Daten-File laden
IDNT	Programm benennen
AUTO	Automatische Befehlsausführung

7.2.1 Direktiven zur Assemblierungskontrolle

SECTION Beginn einer Programmsektion

Syntax: [<Label>] SECTION <Name>[,<Typ>]

Diese Direktive erzeugt eine **neue Sektion** mit der Kennung »Name«, der Programmzähler wird auf 0 gesetzt. Falls »<Name>« bereits verwendet wurde, wird der Programmzähler auf das Ende dieser Sektion gesetzt.

<Name> ist ein String, der optional in Anführungsstriche gesetzt werden kann.

<Typ> muß, wenn angegeben, einer der folgenden sein:

CODE erzeugt eine Sektion relozierbaren Codes

DATA erzeugt eine Sektion initialisierter Daten

BSS erzeugt eine Sektion nicht initialisierter Daten

Um den erzeugten Programmteilen bereits beim Laden einen **Speichertyp** zuzuweisen, können die obigen Typen noch wie folgt erweitert werden:

DATA_C plaziert Daten im Chip-Memory

DATA_F plaziert Daten im Fast-Memory

Aus Kompatibilitätsgründen ist auch die **Extension** »**_P**« vorhanden, die jedoch keinen expliziten Speicher anfordert, also dem Typ ohne die Extension entspricht. Analog können die Erweiterungen auch bei »CODE« und »BSS« verwendet werden. Die explizite Angabe von Fast-Memory ist jedoch nicht unproblematisch, da solche Programme auf Rechnern ohne Fast-Memory nicht laufen. Benutzen Sie diese Angabe nur,

wenn Sie für das Programm unbedingt auf Fast-Memory angewiesen sind.

Für Sektionen, die die **Extension** »_**F**« oder »_**C**« tragen, werden beim Assemblieren automatisch entsprechende Speicherbereiche belegt, damit Sie das Programm unmittelbar testen können. Wenn das nicht automatisch bei jedem Assemblieren eingestellt werden soll, können Sie die Option »AutoAlloc« im Menü »Preferences« ausschalten.

Sie können bis zu 255 Sektionen erzeugen. Als **Default** (Voreinstellung), wenn keine Sektionsdirektive das Programm einleitet, beginnt ASM-ONE mit einer Code-Sektion namens »TEXT«.

Dem **Label**, das der Direktive optional vorangehen darf, wird, wenn vorhanden, die Startadresse der neuen Sektion zugewiesen.

RORG Relativer Programmstart

Syntax: [<Label>] RORG <AbsWert>

Diese Direktive modifiziert den **Programmzähler** so, daß er danach die entsprechende Anzahl von Bytes hinter dem Sektionsbeginn zeigt, vorausgesetzt, daß »AbsExp« nicht kleiner als der aktuelle Offset war.

Dem **Label** wird, wenn vorhanden, der neue Wert des Programmzählers zugewiesen.

ORG Absoluter Programmstart

Syntax: [<Label>] ORG <AbsWert>

Diese Direktive setzt den **Programmzähler** auf den angegebenen absoluten Wert.

WARNUNG: Der Amiga ist ein Multitasking-System, in dem theoretisch alles, außer der »Execbase« an Adresse »4«, auch an beliebigen Adressen liegen kann. Diese Direktive sollte daher nur in begründeten Ausnahmefällen und auf eigenes Risiko eingesetzt werden. Der Vorteil absoluter Adressen ist, daß Sie genau wissen, an welchen Adressen Ihr Programm und Ihre Daten liegen. Viele Hardware-orientierte Spiel- und Demo-Programmierer arbeiten daher mit dieser Direktive.

LOAD Ladeadresse (absolut) setzen

Syntax: [<Label>] LOAD <AbsWert>

Diese Direktive ändert die **Ausgabeadresse** des assemblierten Codes. Das erlaubt Ihnen, ein Programm für eine absolute Adresse zu assemblieren und an eine andere abzulegen. Für weitere Einzelheiten siehe »ORG«.

OFFSET Definiert Offsets

Syntax: [<label>] OFFSET <AbsWert>

Die Direktive »OFFSET« beginnt eine **Offset-Definition.** Ein Beispiel:

```
start   OFFSET  100
dat0    ds.b    1
dat1    ds.b    9
        ENDOFF
```

Dem **Label** »start« wird die Startadresse dieser Offset-Definition zugewiesen. »dat0« erhält den Wert »<AbsWert>« = 100 und »dat1« den Wert »dat0« +1 = 101.

Auch die **Daten** werden hierbei in der aktuellen Sektion abgelegt. In Ihrem Quelltext können Sie die oben definierten Offsets wie folgt verwenden:

```
        LEA     start,a4
        MOVE.B  dat0(a4),d0
        LEA     dat1(a4),a0
.loop   CLR.B   (a0)+
        DBF     D0,.loop
```

Der Zugriff auf Daten wird auf diese Weise schneller und kürzer, da diese Variante für jeden Zugriff auf die Daten zwei Byte weniger Code braucht und zusätzliche vier Byte in der Relozierungstabelle spart. Eine **Offset-Definition** wird durch eine der folgenden Direktiven **beendet**:

```
ENDOFF
SECTION
OFFSET
END
```

ENDOFF Beendet Offset-Tabelle

Syntax: [<Label>] ENDOFF

Siehe »OFFSET«.

END Beendet Programm

Syntax: [<Label>] END

Markiert das **Ende** Ihres Quelltextes. Wenn der Direktive »END« nichts mehr folgt, kann sie auch entfallen. Andere Assembler erwarten die Direktive »END« aber unter Umständen zwingend.

BASEREG Setzt Basisregister

Syntax: [<Label>] BASEREG <label>,An

Definiert ein Register als **Basisregister**. Davon werden die beiden Adressierungsarten

nn(An) und **nn(An,Rn.s)**

beeinflußt. Normalerweise ist »nn« hier eine vorzeichenbehaftete 16-Bit-Ganzzahl.

Wenn »An« aber mit »BASEREG« als Basisregister definiert wurde, wird »nn« als Offset von dieser Adresse angesehen. Das folgende Beispiel

```
           LEA      DataArea,A4
           MOVE.W   D0,DataWord-DataArea(A4)
           ---
DataArea:           dcb.b    100
DataWord:           dc.w     0
```

läßt sich unter Verwendung der Direktive »BASEREG« wie folgt schreiben:

```
           BASEREG   DataArea,A4
           LEA       DataArea,A4
           MOVE.W    D0,DataWord(A4)
           ---
DataArea:  dcb.b     100
DataWord:  dc.w      0
```

Auf diese Weise ist es viel einfacher, ein **Adreßregister** als globalen Zeiger auf einen Datenbereich zu verwenden. Alle Adreßregister stehen zu diesem Zweck (zum Beispiel für verschiedene Datenbereiche) zur Verfügung, allerdings können Sie nach einmaliger Belegung im Quelltext nicht mehr neu definiert werden.

Normalerweise verwendet man eine »**BSS**«-**Sektion**, um Daten abzulegen. Auf diese Weise gewinnen Sie einen schnelleren und kürzeren Zugriff auf die dortigen Daten (»A4« wird auch von C-Compilern normalerweise als Zeiger auf den Datenbereich genutzt).

7.2.2 Direktiven zur Datendefinition

DC Definiert Konstante

Syntax: [<Label>] DC[.Size] <Exp>[,<Exp>[...]]

Diese Direktive leitet eine **Konstantendefinition** ein. Sie kann von einer oder mehreren Zahlen gefolgt sein, die dadurch direkt in den Speicher abgelegt werden. Gültige Größenangaben sind:

.B - Byte
.W - Word ; Wortgröße ist gleichzeitig Default
.L - Long

Wenn als **Größe** »**Byte**« angegeben ist, können Sie auch Strings direkt und ohne trennende Kommas zwischen den Buchstaben angeben:

```
DC.B    'Hallo, wie gehts ?'
```

Bei »**DC.W**« und »**DC.L**« werden ASCII-Werte rechtsbündig in ein Feld der angegebenen Größe übertragen.

Wenn Sie Daten wort- oder langwortweise an einer ungeraden Adresse angeben, erzeugt der Assembler eine Warnung und paßt die Adresse automatisch an die folgende gerade Adresse an.

Um die Warnung zu vermeiden und zu anderen Assemblern kompatibel zu bleiben, die in einer solchen Situation teilweise mit einer Fehlermeldung abbrechen, sollten Sie die Direktive »EVEN« nutzen.

DCB Definiert Konstantenblock

Syntax: [<Label>] DCB[.Size] < AbsWert>,<Wert>

Die Direktive »DCB« erlaubt Ihnen, einen **Block**, der »<AbsWert>« Elemente enthält, zu **definieren**. Jedes der Elemente hat die »<Size>« entsprechende Größe und wird mit dem Wert »<Wert>« initialisiert. Gültige Größen sind:

.B - Byte
.W - Word ; Wortgröße ist gleichzeitig Default
.L - Long

Wenn Sie Daten wort- oder langwortweise an einer ungeraden Adresse angeben, erzeugt der Assembler eine Warnung und paßt die Adresse automatisch an die folgende gerade Adresse an. Um die Warnung zu vermeiden und zu anderen Assemblern kompatibel zu bleiben, die in einer solchen Situation teilweise mit einer Fehlermeldung abbrechen, sollten Sie die Direktive »EVEN« nutzen.

DS Definiert Freiraum für Variablenspeicherung

Syntax: [<Label>] DS[.Size] <AbsWert>

Die Direktive »DS« erlaubt Ihnen, einen **Block**, der »<AbsWert>« Elemente enthält, zu **definieren**. Jedes der Elemente hat die »<Size>« entsprechende Größe, wird jedoch nicht mit einem Wert initialisiert. Gültige Größen sind:

.B - Byte
.W - Word ; Wortgröße ist gleichzeitig Default
.L - Long

Wenn Sie Daten wort- oder langwortweise an einer ungeraden Adresse angeben, erzeugt der Assembler eine Warnung und paßt die Adresse automatisch an die folgende gerade

Adresse an. Um die Warnung zu vermeiden und zu anderen Assemblern kompatibel zu bleiben, sollten Sie die Direktive »EVEN« nutzen.

BLK Block (siehe »DCB«)

Syntax: [<Label>] BLK[.Size] <AbsWert>,<Wert>

Diese Direktive wurde implementiert, um die **Kompatibilität** mit einigen Nicht-Standard-Assemblern zu erhalten. In Programmen, die Sie mit ASM-ONE schreiben, sollten Sie statt dessen »DCB« verwenden.

DR Definiert relativen Wert

Syntax: [<Label>] DR[.Size] <Wert>

Die Direktive »DR« erlaubt Ihnen, **relative Werte** zu **definieren**. Der im Speicher abgelegte Wert ist hierbei

```
<Wert>-*            (* = current address)
```

Die **Größe** dieses Wertes wird nochmals mit der angegebenen Größe verglichen, wobei geprüft wird, ob er sich im zulässigen Wertebereich für eine vorzeichenbehaftete Ganzzahl der angegebenen Größe befindet. Ein Anwendungsbeispiel:

```
JUMP    LEA     DATA(PC),A0
        ADD.W   D0,D0
        ADD.W   D0,A0
        ADD.W   (A0),A0
        JMP     (A0)

DATA    DR.W    ROUTINE_PRINT       ; d0 = 0
        DR.W    ROUTINE_CLEAR       ; d0 = 1
```

Die Routine »JUMP« springt zu der Routine, deren Nummer in »D0« steht.

Gültige Größen sind:

.B	- Byte	
.W	- Word	; Wortgröße ist gleichzeitig Default
.L	- Long	

Wenn Sie Daten wort- oder langwortweise an einer ungeraden Adresse angeben, erzeugt der Assembler eine Warnung und paßt die Adresse automatisch an die folgende gerade Adresse an.

Um die Warnung zu vermeiden und zu anderen Assemblern kompatibel zu bleiben, die in einer solchen Situation teilweise mit einer Fehlermeldung abbrechen, sollten Sie die Direktive »EVEN« nutzen.

7.2.3 Direktiven zur Symboldefinition

EQU Weist dauerhaften Wert zu

Syntax: <Label> EQU <Wert>

Diese Direktive **weist** dem **Label** »<Label>« einen **Wert zu**. »<Label>« kann nun als konstanter Wert in anderen Termen verwendet werden.

Diese Direktive entspricht dem Statement »=«.

SET Weist temporären Wert zu

Syntax: <Label> SET <Wert>

Diese Direktive **weist** einem **Label** ebenfalls einen **Wert zu**. Dieses Label kann nun ebenso wie ein mit »EQU« definiertes als Symbol in Gleichungen verwendet, jedoch anders als ein solches auch neu definiert werden.

Anmerkung: Sie sollten keine Vorwärtsreferenzen zu einem »SET«-Symbol verwenden.

EQUR Weist Register zu

Syntax: <Label> EQUR <Rn>

Diese Direktive erlaubt, einem **Register** einen **symbolischen Namen** zuzuweisen. Sie können dies verwenden, wenn Sie kritische Bereiche Ihres Programms bearbeiten, in denen Sie viele Register verwenden. Zum Beispiel:

```
Eg.  BitPlane1  EQUR     A3
                MOVE.L   D0,(BitPlane1)+
```

Der Name »BitPlane1« kann nicht neu definiert werden, es ist aber möglich, einem Register mehrere Namen zu geben.

Anmerkung: Sie haben hierbei nur die Daten- und Adreßregister zur Verfügung.

REG Weist Registerliste zu

Syntax: <Label> REG <Liste>

Diese Direktive wird ähnlich wie die Direktive »EQUR« angewendet, doch können Sie hiermit einer **Registerliste** einen **Namen zuweisen**, wie Sie beim Befehl »MOVEM« verwendet wird.

```
Eg.     AllRegs REG         D0-A6
                MOVEM.L     AllRegs,-(A7)
```

Der Name »AllRegs« kann auch hier nicht neu definiert werden, es ist aber ebenfalls möglich, derselben Registerliste mehrere Namen zu geben. **Anmerkung**: Sie haben auch hierbei nur die Daten- und Adreßregister zur Verfügung.

RS Weist relativen Wert zu

Syntax: [<Label>] RS.[size] <AbsWert>

Die Direktive »RS« weist dem **Label** den derzeitigen **Wert** des Assembler-internen **RS-Zählers** zu. Anschließend wird der Wert zu »RS« addiert. Benutzen Sie, um »RS« zu löschen, die Direktive »RSRESET«. Die Direktive »RS« gibt Ihnen eine schnelle und einfache Methode, **Listen** von **Offsets** zu **definieren**, beispielsweise Library-Offsets:

```
        RSRESET
        RS.B    -30         ; Startwert
Open    RS.B    -6
Close   RS.B    -6
Read    RS.B    -6
Write   RS.B    -6
Input   RS.B    -6
Output  RS.B    -6
```

RSRESET Setzt Zähler für relative Offsets zurück

Syntax: [<Label>] RSRESET

Eine genaue Beschreibung finden Sie bei der Direktive »RS«.

RSSET Setzt Zähler für relative Offsets

Syntax: [<Label>] RSSET <AbsWert>

Anstatt »RS« nur zu löschen, können Sie gleichzeitig auch einen neuen Startwert zuweisen. Dies entspricht der Befehlsfolge

```
RSRESET
RS.B    <AbsWert>
```

Eine genaue Beschreibung finden Sie bei der Direktive »RS«.

7.2.4 Makro-Direktiven

MACRO Beginnt eine Makro-Definition

Syntax: <Label> MACRO

Mit dieser Direktive **beginnt** eine **Makro-Definition**, wobei »<Label>« in diesem Falle der Name des Makros ist. Benutzen Sie die Direktive »ENDM«, um eine Makro-Definition zu beenden.

Makros werden verwendet, um einer kurzen Code-Sequenz einen Namen zuzuweisen.

Es ist möglich, ein **Makro** mit Operanden, gleichsam **als Parameter**, aufzurufen. Ein Operand wird in einem Makro mit einem Backslash »\« bezeichnet, gefolgt von einer Zahl zwi-

schen 1 und 9, die die Nummer des Operanden angibt. Ein von einem Klammeraffen »@« gefolgter Backslash wird beim Assemblieren durch die Anzahl der bisherigen Aufrufe dieses Makros ersetzt, so daß Labels im Makro, denen ein »\@« angehängt oder vorweggestellt wird, beim zweiten Aufruf eines Makros keinen Double-Symbol-Fehler hervorrufen.

Ein Beispiel für ein einfaches Makro:

```
CoordsXYZ  MACRO
           DC.W     \1,\2,\3
           ENDM
```

In Ihrem Programm schreiben Sie dann einfach:

```
CoordsXYZ  10,10,10
CoordsXYZ  10,-10,10
---
---
CoordsXYZ  -10,10,10
```

Falls Sie die Koordinaten für eine andere Anwendung mit einem Faktor, beispielsweise 16, multiplizieren müssen, schreiben Sie das Makro einfach um:

```
CoordsXYZ  MACRO
           DC.W     \1*16,\2*16,\3*16
           ENDM
```

Sie können keine Makros innerhalb eines Makros definieren, aber Sie können andere Makros aufrufen oder rekursiv dasselbe Makro erneut aufrufen. Um solche Makros zu beenden, müssen Sie »MEXIT« oder »CMEXIT« verwenden. Die Grenze für die Verschachtelung von Makros liegt bei 25 Rekursionen.

NARG Spezielles Symbol (Zahl der Parameter)

Syntax: NARG

Dieses spezielle **Symbol** bekommt beim Makro-Aufruf die Zahl der übergebenen **Parameter** zugewiesen. Außerhalb eines Makros hat »NARG« den Wert 0.

ENDM Beendet Makro-Definition

Syntax: ENDM

Beendet eine **Makro-Definition**. Siehe auch bei »MACRO«, »MEXIT« und »CMEXIT«.

MEXIT Verläßt Makro

Syntax: MEXIT

Beendet ein **Makro vorzeitig**. Wenn der Assembler diese Direktive erreicht, wird das Makro beendet, auch wenn das reguläre Ende noch nicht erreicht war.

CMEXIT Verläßt Makro, wenn Rekursionstiefe erreicht

Syntax: CMEXIT <AbsWert>

Beendet ein **Makro**, wenn die derzeitige **Rekursionstiefe** »<AbsWert>« entspricht. Wenn diese Tiefe erreicht ist, wird das Makro beendet, auch wenn das reguläre Ende noch nicht erreicht war.

REPT Beginnt Code-Wiederholung

Syntax: REPT <AbsWert>

Leitet die **wiederholte Assemblierung** eines Blocks ein. Sie können dies verwenden, um einen oder mehrere Befehle mehrfach zu wiederholen. Das Ende des Blocks wird durch »ENDR« markiert.

Wenn Sie beispielsweise 10 Byte sehr schnell kopieren wollen, könnten Sie das wie folgt tun:

```
REPT 10
MOVE.B   (A0)+,(A1)+
ENDR
```

ENDR Beendet Code-Wiederholung

Syntax: ENDR

Beendet die **wiederholte Assemblierung**; s. auch »REPT«.

7.2.5 Bedingte Assemblierung

CNOP Bedingtes NOP zur Ausrichtung an Wort- oder Langwortgrenzen

Syntax: [<Label>] CNOP <AbsWert1>,<AbsWert2>

Diese Direktive **richtet** die **Adresse** auf eine durch »<AbsWert2>« teilbare Adresse **aus** und addiert dann »<AbsExp1>« zu dieser Adresse. Zum Beispiel:

```
eg.     CNOP     0,4
```

richtet auf eine Langwortgrenze aus.

```
eg.     CNOP     2,4
```

richtet auf ein Wort hinter einer Langwortgrenze aus.

»**<Label>**« wird, wenn vorhanden, der neue Wert des Programmzählers nach dem »CNOP« zugewiesen.

Anmerkung: »CNOP« führt keine Initialisierung aus.

EVEN Erzwingt Fortsetzung auf gerader Adresse

Syntax: [<Label>] EVEN

Diese Direktive hat dieselben Auswirkungen wie

```
CNOP   0,2
```

»**<Label>**« wird, wenn vorhanden, der neue Wert des Programmzählers nach dem »EVEN« zugewiesen.

Anmerkung: »EVEN« führt keine Initialisierung aus.

ODD Erzwingt ungerade Adresse

Syntax: [<Label>] ODD

Kann auch ersetzt werden durch

```
CNOP 1,2
```

»ODD« **ändert** einen bereits **ungeraden Programmzähler nicht**. (Die Variante »CNOP« würde in diesem Falle zwei Byte addieren.)

»**<Label>**« wird, wenn vorhanden, der neue Wert des Programmzählers nach dem »ODD« zugewiesen.

Anmerkung: »ODD« führt keine Initialisierung aus.

IFEQ	Assembliert, wenn Term	**= 0**	(Equal)
IFNE	Assembliert, wenn Term	**<> 0**	(Not Equal)
IFGT	Assembliert, wenn Term	**> 0**	(Greater Than)
IFGE	Assembliert, wenn Term	**>= 0**	(Greater or Even)
IFLT	Assembliert, wenn Term	**< 0**	(Less Than)
IFLE	Assembliert, wenn Term	**<= 0**	(Less or Even)

Die obigen Varianten können auch ersetzt werden durch das allgemeinere

IF Assembliert, wenn logisch wahr

Syntax: IF <boolean>

Ein solcher boolescher Wahrheitswert besteht aus zwei Termen, die durch einen der folgenden Vergleichsoperatoren in Relation gebracht werden:

=, >, <, >=, <=, <>

Wenn eine »IF(cc)«-Direktive nicht zutrifft, wird der folgende Code übersprungen, bis ein »ENDC« oder ein Else-Statement erreicht wird.

»IF(cc)«-Direktiven können bis zu 25fach verschachtelt werden.

IFC Assembliert, wenn **Strings identisch** sind

Syntax: IFC <string>,<string>

Siehe auch die allgemeine Erläuterung zu »IF«.

IFNC Assembliert, wenn **Strings nicht identisch** sind

Syntax: IFNC <string>,<string>

Siehe auch die allgemeine Erläuterung zu »IF«.

IFD Assembliert, wenn ein **Symbol definiert** ist

Syntax: IFD <Wert>

Siehe auch die allgemeine Erläuterung zu »IF«.

IFND Assembliert, wenn ein **Symbol nicht definiert** ist

Syntax: IFND <Wert>

Siehe auch die allgemeine Erläuterung zu »IF«.

IFB Assembliert, wenn **leer**

Syntax: IFB <symbol>

Diese Instruktion testet das folgende »<Symbol>«, wenn dort nichts steht, ist die Bedingung »leer« erfüllt. Dieser Befehl findet Anwendung, um herauszufinden, ob ein Makro-Parameter vorhanden ist oder nicht.

Siehe auch die allgemeine Erläuterung zu »IF«.

IFNB Assembliert, wenn **nicht leer**

Syntax: IFNB <symbol>

Diese Instruktion testet das folgende »<Symbol>«, wenn dort etwas steht, ist die Bedingung »nicht leer« erfüllt. Dieser Befehl findet Anwendung, um herauszufinden, ob ein Makro-Parameter vorhanden ist oder nicht.

Siehe auch die allgemeine Erläuterung zu »IF«.

IF1 Assembliert, wenn in »Pass1«

Syntax: IF1

Der Assembler arbeitet den folgenden Block **nur** in »**Pass1**« ab.

Siehe auch die allgemeine Erläuterung zu »IF«.

IF2 Assembliert, wenn in »Pass2«

Syntax: IF2

Der Assembler arbeitet den folgenden Block **nur** in »**Pass2**« ab.

Siehe auch die allgemeine Erläuterung zu »IF«.

ELSE Alternativer Code, wenn Bedingung nicht erfüllt

Syntax: ELSE

Siehe auch die allgemeine Erläuterung zu »IF«.

ENDC Beendet »IF«-Block

Syntax: ENDC

Siehe auch die allgemeine Erläuterung zu »IF«.

7.2.6 Ausgabekontrolle

PAGE Beginnt neue Seite im Listfile

Syntax: PAGE

Beginnt eine **neue Seite** und aktiviert die Seitentrennung.

NOPAGE Schaltet Seitentrennung ab

Syntax: NOPAGE

LIST Aktiviert Listing

Syntax: LIST

Beginnt mit der **Erzeugung** des **Listfiles**.

NOLIST (NOL) Beendet Listing

Syntax: NOLIST (or NOL)

Beendet Erzeugung des **Listings**.

LLEN Setzt Zeilenlänge

Syntax: LLEN <AbsWert>

Setzt die **Zeilenlänge** (zulässige Werte sind 60 bis 132).

PLEN Setzt Seitenlänge (Zeilenzahl)

Syntax: PLEN <AbsWert>

Setzt die **Seitenlänge** (zulässige Werte sind 20 bis 100).

SPC Erzeugt n leere Zeilen

Syntax: SPC <AbsWert>

Fügt n **leere Zeilen** in das Listfile ein.

TTL Setzt Programmtitel

Syntax: TTL <string>

Setzt den **Programmtitel** (für Listfile). Um dem Programm-Modul einen Namen zu geben, verwenden Sie »IDNT«.

FAIL Generiert Assembler-Fehler

Syntax: FAIL

Erzeugt einen **Fehler**. Der Fehler taucht auch im Listing auf.

PRINTT Ausgabe eines Strings auf den Bildschirm

Syntax: PRINTT <string>[,<string>,...]

Gibt während des Assemblierens einen **String** auf den **Bildschirm** aus. Sie können diese Anweisung benutzen, um anderen, die Ihren Quellcode assemblieren, Informationen oder Hinweise zu geben. Sie können sie aber auch als Gedächtnisstütze für sich selbst benutzen, etwa in dieser Art:

```
PRINTT "Diese Zeile wurde assembliert"
```

PRINTV Ausgabe eines Werts auf den Bilschirm

Syntax: PRINTV <Ausdruck>[,<Ausdruck>,...]

Ausgabe eines **Werts** auf den **Bildschirm** während des Assemblierens. Diese Anweisung läßt sich sehr gut einsetzen, um zu überprüfen, ob während des Assembliervorgangs alles korrekt abläuft. Besonders nützlich ist dies, wenn Sie bedingtes Assemblieren oder »REPT«-Statements benutzen.

```
PRINTV ScreenAddress, ScreenWidth
```

7.2.7 Externe Symbole

XDEF Externe Definition

Syntax: XDEF <Label>[,<Label>..]

Definiert ein **Symbol** als »**extern**«. Jedes Symbol darf nun auch aus anderen Modulen referenziert werden, auch aus Hochsprachenmodulen.

Ein Beispiel für die Anwendung dieser Direktive finden Sie unter »XREF«.

Hinweis: Programme, die »XDEF« oder »XREF« verwenden, können nicht direkt ausgeführt werden.

XREF Externe Referenz

Syntax: XREF <Label>[,<Label>..]

Importiert Symbole, die in anderen Modulen definiert wurden. Jedes Symbol kann nun ebenso wie ein im Modul definiertes relatives Symbol behandelt werden.

Ein Beispiel für die Anwendung dieser Direktive im folgenden.

Geben Sie das Beispielprogramm ein, assemblieren und speichern Sie es als Objektmodul.

```
        XDEF    ClearScreen
ClearScreen
        CLR.L   Screen
        RTS
Screen  DC.L    0
```

Geben Sie dann das folgende Beispielprogramm ein, assemblieren und speichern Sie es ebenfalls.

```
         XREF   ClearScreen
GoClear  JMP    ClearScreen
```

Sie haben nun zwei Objektmodule. Verwenden Sie einen Linker, um diese beiden Objektmodule zu einer ausführbaren Datei zusammenzubinden.

Hinweis: Programme, die »XDEF« oder »XREF« verwenden, können nicht direkt ausgeführt werden.

7.2.8 Allgemeine Direktiven

JUMPPTR Setzt Startpointer

Syntax: JUMPPTR <Label>

Diese Direktive **setzt** die **Startadresse** für das Debuggen eines Programms. Normalerweise ist die Startadresse die Adresse des ersten Befehls in der ersten nicht leeren Programm-Sektion. An diese Adresse springt der Assembler, wenn Sie die Befehle »J« oder »G« ohne Adreßangabe verwenden.

INCBIN Binär-File einfügen

Syntax: INCBIN <Filename>[,<AbsWert>]

Fügt binäre Daten in Ihr Programm ein. Diese Daten werden beim Assemblieren nachgeladen. Ein Beispiel:

```
START:
        INCBIN  "datafile"
END:
```

Wenn »datafile« ein binäres File mit einer Länge von 1000 Byte ist, wird das Label »END« einen Wert annehmen, der genau 1000 Byte höher ist als der von »START«, und »datafile« wird an die Adresse »START« geladen.

INCLUDE Sourcefile einfügen

Syntax: INCLUDE <Filename>

Fügt zusätzlichen Code in Ihren Quelltext **ein**. Include-Dateien enthalten normalerweise Konstantendefinitionen wie Library-Offsets oder symbolische Parameter, können aber prinzipiell alle Arten von Code enthalten.

Sie nutzen Include-Dateien, indem Sie einen Quelltext erstellen, der all Ihre Definitionen (wie Konstanten, Makros etc.) enthält.

Diesen fügen Sie dann am Beginn Ihres Programms mit »INCLUDE« ein. So können Sie alle Definitionen in der Include-Datei so nutzen, als wenn Sie sie tatsächlich am Beginn Ihres Programms eingefügt hätten.

Ein Beispiel für diese Methode:

```
INCDIR   DF0:INCLUDE\   ; Gibt den Pfad
                        ; an, in dem sich
                        ; Ihre Includes
                        ; befinden
INCLUDE  exec/exec.i    ; Der Name der
                        : einzufügenden Datei
```

Sie können **mehrere Include-Aufrufe** hintereinander vornehmen und auch aus Include-Dateien heraus andere Include-Dateien einbinden. Die maximale Verschachtelungstiefe ist 5.

Um die Assemblierung zu beschleunigen, werden Include-Dateien nach dem erstmaligen Laden im Speicher gehalten. Dadurch müssen sie nicht bei jedem Assemblieren neu nachgeladen werden.

Um den dafür benutzten **Speicher** manuell wieder **freizugeben**, können Sie das Kommando »ZI« (Include-Speicher löschen) verwenden.

INCDIR Pfad für Includes angeben

Syntax: INCDIR <Pfad>

Setzt den Pfad, unter dem nach den Include-Dateien gesucht wird. Default ist

```
INCDIR   "Df0:Includes/"
```

>EXTERN Daten-File laden

Syntax: >EXTERN [<Num>,]<Filename>,<LadeAdr>[,<Länge>]

Definiert eine **Referenz** auf ein externes File mit dem angegebenen File-Namen. Die Datei wird an die Adresse »<LadeAdr>« geladen. Sofern »<Länge>« angegeben wurde, werden maximal »<Länge>« Byte geladen.

»EXTERN«-Dateien werden nachgeladen, wenn Sie den Befehl »E« eingeben. Wenn Sie in Ihren »EXTERN«-Aufrufen den Parameter »<Num>« mit angeben, haben Sie die Möglichkeit, nur bestimmte »EXTERN«-Files zu laden, indem Sie »E<Num>« verwenden.

IDNT Programm benennen

Syntax: IDNT <Name>

Ein **Programm**, das mehr als eine Sektion enthält, muß mit einem **Namen** versehen werden. Normalerweise wird ein Leer-String zugewiesen, aber Sie können dem Programm mit »IDNT« auch einen speziellen Namen geben.

AUTO Automatische Befehlsausführung

Syntax: AUTO <Befehl>[\<Befehl>..]

»AUTO« erlaubt Ihnen, einen oder mehrere **Befehle fest einzustellen**, zum Beispiel

```
AUTO    E\
```

wenn Sie alle externen Files automatisch bei jeder Assemblierung einladen wollen.

Der Backslash »\« bedeutet hier »Return«.

8 Der Debugger

Der Debugger von ASM-ONE bietet Ihnen neben den Funktionen eines normalen Debuggers auch die Möglicheiten eines Source-Level-Debuggers. Das bedeutet, daß Sie Ihren Quelltext quasi direkt ausführen und die Ausführung zeilenweise verfolgen können. Sie sehen Ihren Quelltext im Debugger exakt so wie im Editor, mit Zeilennummern, Kommentaren etc.

Sie können jedoch gleichzeitig alle Registerinhalte und beliebige Speicherstellen unter Kontrolle behalten. Wenn Sie ein Programm debuggen, von dem Sie keinen Quelltext haben, können Sie es im Einzelschrittmodus in einem normalen Disassembler-Fenster abarbeiten lassen. Auch hier stehen Ihnen alle Befehle des Source-Level-Debuggers zur Verfügung. Sie können sogar beim Debuggen eines Programms zwischen beiden Modi wechseln.

Wenn Sie sich im Debugger befinden, stehen Ihnen alle Funktionen zur Verfügung, die im Menü »Debug Funct« aufgelistet sind. Die Funktionen sind folgende:

Step One

Führt einen **einzelnen Programmschritt** aus, indem auf den folgenden Befehl ein Breakpoint gesetzt wird. **Subroutinen** werden so **nicht** im Einzelschrittmodus ausgeführt, sondern in einem Stück abgearbeitet. Wenn Sie auch eine Subroutine im Einzelschrittmodus verfolgen wollen, können Sie das Kommando »Enter« benutzen.

Die **bedingten Verzweigungen** »Bcc«, »DBcc« und der Befehl »RTS« werden im Trace-Modus abgearbeitet. Daraus folgt, daß Sie diese Befehle nicht mit dem Befehl »Enter« abarbeiten müssen, um zu verfolgen, wohin sie verzweigen. »BRA« und »DBRA« werden hier wie bedingte Verzweigungen behandelt. Der Befehl »JMP« wird speziell behandelt. Er wird ebenfalls in einem Schritt abgearbeitet, die Return-Adresse wird jedoch vom Stack geholt und als Breakpoint verwendet.

Dies kann ein Problem verursachen, wenn vorher **Parameter** auf den **Stack** geschoben wurden, die von der mit »JMP« angesprungenen Routine wieder heruntergenommen werden. (Sie finden diese Vorgehensweise häufig in Hochsprachenprogrammen, die Parameter nicht in Registern, sondern über den Stack übergeben.) In diesem Falle ist das zuletzt auf den Stapel geschobene Langwort nämlich nicht die Return-Adresse. Um eine solche Konstruktion zu tracen, müssen Sie den Breakpoint selbst setzen oder den Befehl mit »Enter« abarbeiten lassen.

Enter

Dieser Befehl wird verwendet, um eine **Subroutine**, die normalerweise an einem Stück abgearbeitet werden würde, dennoch im **Einzelschrittmodus** abzuarbeiten. Dies sind Routinen, die durch einen der folgenden Befehle gestartet werden: »BSR«, »JSR« oder »JMP«.

Wenn Sie im Modus »ShowSource« arbeiten (dies ist der Default-Modus, in dem Sie Ihren Quelltext beim Tracen mitverfolgen können), ist es nicht empfehlenswert, eine **Library-Routine** auf diese Weise zu verfolgen, da diese Adresse außerhalb Ihres Quelltextes liegt.

Ist dies doch der Fall, können Sie nachträglich einen Breakpoint setzen und den Rest der Library-Routine in einem Stück abarbeiten lassen oder den Modus »ShowSource« ausschalten und den Rest der Routine anhand des normalen Disassembler-Listings verfolgen. Da es nicht möglich ist, einen Breakpoint im ROM zu setzen, müssen Sie hier **immer** das Kommando »Enter« nutzen.

Step n

Erlaubt es Ihnen, **n Befehle** in einem Schritt abzuarbeiten. Dies kann ein wenig unübersichtlich werden, weil es schwer vorherzusehen ist, wo das Programm beispielsweise nach 143 Schritten sein wird. Der bessere Weg ist normalerweise, einen Breakpoint zu setzen.

Edit Regs

Sie können während der Analyse eines Programms im Einzelschrittmodus die Inhalte sämtlicher **Register** nach Belieben **ändern**. Natürlich greifen Sie dadurch in den Programmablauf ein, so daß die Ergebnisse des Programms nicht mehr denen entsprechen, die es normalerweise ausgegeben hätte. Doch es ist manchmal angenehmer, einen falschen Wert erst einmal nur zu editieren, anstatt den Fehler sofort zu korrigieren und das Programm deswegen neu assemblieren zu müssen.

AddWatch

Hilft Ihnen bei der **Beobachtung eines Speicherbereiches.** Sowohl direkte Adressen als auch Register und Symbole können dafür genutzt werden. Folgende Angaben sind für den zu beobachtenden Bereich zulässig:

```
BUFFER          A0+D0+10
BUFFER+2+D1
etc.
```

Dieser Zeiger kann wahlweise auf die folgenden Datentypen zeigen:

Ascii, String, Hex, Decimal, Binary, Pointer

Ist der Typ des Pointers spezifiziert, sind folgende Typen erlaubt:

```
dc.l    ; Absolut Lang (32 Bits)

dc.w    ; Absolut Wort (16 Bits)

dr.l    ; PC relativ Lang (32 Bits)

dr.w    ; PC relativ Wort (16 Bits)
```

Dieser Zeiger kann wiederum auf alle der obigen Werte zeigen, ausgenommen einen weiteren Pointer.

DelWatch

Jedesmal wenn Sie eine neue **Speicheradresse** mit »AddWatch« spezifizieren, wird deren Name im **Menü** »**DelWatch**« abgelegt. Es ist so möglich, ein einzelnes Element wieder aus der »AddWatch«-Liste zu entfernen, indem Sie das zu entfernende Element einfach anwählen.

ZapWatch

Arbeitet im Prinzip wie »DelWatch«, **löscht** jedoch nicht nur ein Element, sondern **alle Elemente** aus der »Watch«-Liste.

JumpAddr

Springt zu einer **absoluten Adresse**, die Sie in der Menüzeile eingeben können. Diese absolute Adresse kann dabei ein Label, ein Register oder sogar »A0+4« oder »A0+D0+2« etc. sein.

JumpMark

Die **aktuelle Monitorzeile** wird **farbig hinterlegt** und kann mit den Cursor-Tasten hoch und runter bewegt werden. Durch »Return« springen Sie zu der markierten Zeile. »Esc« macht den Sprung wieder rückgängig.

B.P. Addr

Setzt einen **Breakpoint** auf eine absolute Adresse, die Sie auch hier wieder in die Menüzeile eingeben.

B.P. Mark

Ihre **derzeitige Zeile wird hinterlegt**, Sie können diese Zeile nun mit den Cursor-Tasten bewegen. Drücken Sie auch hier »Return«, um den Breakpoint auf die markierte Zeile zu setzen, oder »Esc«, um den Vorgang abzubrechen.

Zap B.P.

Sie können insgesamt 16 verschiedene **Breakpoints setzen**. Jedesmal wenn das Programm an einem Breakpoint unterbrochen wird, wird dieser Breakpoint wieder aus der Liste entfernt. Dieses Kommando ermöglicht Ihnen das Löschen aller Breakpoints.

Zusätzliche Flags:

DisAssem - Zeigt eine disassemblierte Version der Zeile, an der sich der Programmzähler gerade befindet

ShowSource - Zeigt den Quelltext

Um die Arbeitsweise dieser Befehle einmal auszuprobieren, sollten Sie das Programm »BinaryConv.S« aus dem Beispielverzeichnis laden.

9 Der Monitor

Der Monitor ist im Unterschied zum Debugger nicht dazu gedacht, Programme ablaufen zu lassen. Mit dem Monitor haben Sie die Möglichkeit, **Speicherinhalte** anzusehen und zu **editieren**. Der Monitor verhält sich dabei wie ein Editor, Sie können den Speicher des Amiga durchlaufen und dessen Inhalte direkt durch Überschreiben ändern.

Mit dem Monitor können Sie Speicherinhalte auf drei verschiedene Arten betrachten:

1. **Disassembliert**
2. **In hexadezimaler Darstellung**
3. **In ASCII-Darstellung**

Jeder dieser Modi hat dieselben Grundfunktionen, bietet aber zusätzlich noch spezielle Möglichkeiten.

Im **Disassembliermodus** sind das:

Alle alphanumerischen Tasten

Aktivieren den Zeilen-Assembler. In diesem Modus haben Sie Zugriff auf alle Editiermöglichkeiten einer Zeileneingabe. Wenn der eingegebene Befehl unzulässig ist, wird keine Änderung vorgenommen.

Nicht-alphanumerische Tasten (z. B. Cursor-Tasten)

Das Drücken der nicht-alphanumerischen Tasten aktiviert den Zeilenassembler-Modus. Anstatt aber in einer freien Reihe zu beginnen, wird die aktuelle Zeile in den Zeilenpuffer gelegt. Das vereinfacht die Änderung von bestehendem Code.

Haupttasten:

Ctrl-Shift-B: Ändert Hex-Ausgabe auf Byte-Größe

Ctrl-Shift-W: Ändert Hex-Ausgabe auf Wort-Größe

Ctrl-Shift-L: Ändert Hex-Ausgabe auf Langwort-Größe

»**DEL**«

Fügt eine »NOP«-Instruktion an der angegebenen Adresse ein und bewegt die aktuelle Position auf die nächste Adresse.

Im **Hexadezimal- und im ASCII-Anzeigemodus** können Sie folgende Tasten verwenden:

Cursor-Tasten links/rechts mit »**Shift**«

Springen an den Anfang beziehungsweise ans Ende der aktuellen Zeile.

Cursor-Tasten links/rechts mit »**Alt**«:

Scrollen den Screen nach links beziehungsweise rechts.

Nun folgen die **allgemeinen Kommandos**, die Ihnen in allen drei Modi gleichermaßen zur Verfügung stehen (Sie finden diese Befehle auch im Monitor-Menü):

DisAssem

Wechselt in den **Disassembliermodus**. Es ist möglich, unter den Modi zu wechseln, ohne den Monitor zu verlassen. Falls Sie in einem Disassembler-Listing auf einen Bereich stoßen sollten, der nicht wie Maschinencode aussieht, können Sie direkt in den ASCII- oder Hex-Modus wechseln, um den Bereich genauer zu untersuchen, und ebenso wieder zurück.

HexDump

Wechselt in den **Hexadezimal-Modus**.

AsciiDump

Wechselt in den **ASCII-Modus**.

Jump Addr

Springt an eine **Adresse**. Geben Sie die gewünschte Adresse in der Eingabezeile ein. Wenn die Adresse ungültig ist, wird der Sprung nicht ausgeführt. Die letzte Adresse wird in einem Puffer (siehe »Last Addr«) gespeichert, so daß Sie problemlos wieder an die Adresse zurückspringen können, von der Sie ausgegangen sind.

Last Addr

Kehrt zur **letzten Adresse zurück**, die Sie betrachteten. Der Puffer des Monitors kann bis zu 16 Positionen speichern.

Mark 1..3

Markiert eine **Position**, an die Sie später nochmals zurückkehren wollen. Sie können Marken benutzen, um zwischen verschiedenen Speicherbereichen hin- und herzuspringen.

Jump 1..3

Springt an eine **gespeicherte Position**. Eine Anwendung dieses Befehls ist die ständige Überwachung eines sich ändernden Speicherbereiches. Setzen Sie dazu eine Marke an den Anfang des Bereiches, und benutzen Sie den Befehl »Jump«, um zu dieser Marke zu springen und eine Erneuerung der Bildschirmanzeige auszulösen.

QuickJump

Wenn Sie Sprünge nutzen wollen, müssen Sie normalerweise die Adresse angeben. Diese Funktion arbeitet wie folgt:

Beim Disassemblieren:

Suchen Sie das als nächstes erreichbare Langwort, und bringen Sie es auf die »Current Line«, so daß es invers dargestellt wird. Ist kein Langwort im Text erreichbar, wird kein Sprung ausgeführt.

ACHTUNG: Es wird nicht geprüft, ob die Speicheradresse der Langwortadresse legal ist. Ist dies nicht der Fall, kann dies einen Absturz zur Folge haben.

In Hexadezimal- oder ASCII-Form:

Das vom Cursor übernommene, in der am nächsten gelegenen geraden Adresse liegende Langwort wird in das Gadget »Jump Addr« kopiert. Wird dies akzeptiert, braucht nur mit der Taste »RETURN» bestätigt zu werden.

esc

Verläßt den **Monitor** und kehrt in die Befehlszeile zurück.

Zusätzliche Flags:

OnlyASCII. -Unterdrückt bei der Anzeige im ASCII-Modus alle Nicht-ASCII-Zeichen. Normalerweise wird der gesamte Zeichensatz zur Anzeige genutzt.

WARNUNG:

Mit dem Monitor können Sie sämtliche Speicherbereiche beobachten und ändern, auch wenn diese Speicherbereiche im System des Amiga nicht als beschreibbarer Speicher verfügbar sind.

Zugriffe auf nicht existierende Speicherbereiche oder den I/O-Bereich können das System zum Absturz bringen.

Auch Modifikationen in Speicherbereichen, die vom Betriebssystem belegt sind, können zum Absturz oder zu Fehlfunktionen des Computers führen. Sie gehen dabei das Risiko ein, den im Editor befindlichen Quelltext zu verlieren. **Speichern Sie Ihren Quelltext also ab, bevor Sie in den Monitor wechseln.**

10. Der Motorola MC68000

Der MC68000 ist ein 16/32-Bit-Prozessor. Das bedeutet, daß der externe Datenbus nur 16 Bit breit ausgelegt ist, intern aber mit vollen 32 Bit gearbeitet wird. Der Adreßbus ist beim MC68000 24 Bit breit ausgelegt, so daß der maximale Adreßbereich 2^{24} Bytes = 16 MByte umfaßt.

Da die niederwertigste Adreßleitung nicht herausgeführt ist, greift der MC68000 immer nur wortweise und nur an geraden Adressen auf den Speicher zu.

Der Prozessor hat folgende Register:

Datenregister	D0 bis D7	8, 16, 32 Bit
Adreßregister	A0 bis A7	16, 32 Bit
Benutzer-Stackpointer	USP	32 Bit
Supervisor-Stackpointer	SSP	32 Bit
Programmzähler	PC	24 Bit
Statusregister	SR	16 Bit
Bedingungscoderegister	CR	8 Bit

Datenregister (d0-d7):

Diese Register können sowohl als Zähler als auch als Konstanten verwendet werden. Sie können mit diesen Registern auch rechnen, wobei logische und arithmetische Operationen zur Verfügung stehen.

Adreßregister (a0-a6):

Diese Register können Sie als Zeiger und als Konstanten nutzen. Sie können mit Adreßregistern ebenfalls rechnen, wobei Sie jedoch auf die arithmetischen Operationen beschränkt sind.

Stapelzeiger (A7, USP, SSP):

Es gibt zwei verschiedene Stapelzeiger, getrennt für den Benutzer- und den Supervisor-Stack. Der Stapel speichert beispielsweise die Rücksprungadresse aus einer Unterroutine und kann ebenfalls genutzt werden, um Register zu sichern.

Ein Bit im Statusregister selektiert, welcher der beiden Stapel benutzt wird. Normalerweise ist das der Benutzer-Stack, aber in Interrupts und anderen Ausnahmeroutinen wird der Supervisor-Stack verwendet. Der derzeitige Stackpointer wird in »A7« gesichert; es ist also nicht möglich, dieses Register in eigenen Routinen zu verwenden.

Das folgende kleine Beispielprogramm gibt die Return-Adresse einer Subroutine in »D0« aus:

```
START     BSR      SUBROUT
CONT      RTS
SUBROUT   MOVE.L   (A7),D0
          RTS
```

Assemblieren Sie dieses Beispiel, und starten Sie es mit »J«. Betrachten Sie »D0« nach dem Start. Sehen Sie sich den Wert des Labels »CONT« mit »?CONT« an. Haben Sie dieselbe Adresse bekommen?

Sie können auf dem Stapel auch Register sichern, wenn Sie deren Inhalt aus irgendwelchen Gründen nicht einfach löschen dürfen:

```
START   MOVEM.L   D0-A6,-(A7)
        NOP             ; Hier können Sie nun
                        ; Ihre Routine einfügen
        MOVEM.L   (A7)+,D0-A6 RTS
```

Status- und Bedingungscoderegister (SR, CCR):

Das »CCR«-Register bezeichnet nur die unteren 8 Bit des Statusregisters, so daß die Beschreibung dieser beiden Register sich zusammenfassen läßt.

Register SR und CCR:

Bit	Name	Bedeutung
0	C, Carry	Gesetzt, wenn Ergebnis Wertebereich überschritt
1	V, Overflow	Gesetzt, wenn Überlauf (bei vorzeichenbehafteten Zahlen)
2	Z, Zero	Gesetzt, wenn das Ergebnis null war
3	N, Negative	Gesetzt, wenn das Ergebnis negativ war
4	X, Extended	Vergleichbar dem Carry-Bit, jedoch anders beeinflußt; für mehrfach-genaue Operationen
5 - 7	unbenutzt	

Das höherwertige Byte von SR:

Bit	Name	Bedeutung
8 - 10	I0 - I2	Interrupt-Level (3-Bit-Wert)
11 - 12	unbenutzt	
13	S, Supervisor	Gesetzt, wenn im Supervisor-Modus; erlaubt die Ausführung aller privilegierten Befehle und Verwendung des SSP-Stacks
14	unbenutzt	
15	T, Trace	Gesetzt, wenn Trace-Modus aktiv; nach jedem einzelnen Befehl wird eine Trace-Exception ausgelöst, so daß immer nur ein Befehl ausgeführt wird

10.1 Der Befehlssatz

Zunächst eine Beschreibung der verwendeten Abkürzungen:

Label	bezeichnet ein Symbol oder eine Adresse
Rn	Adreß- oder Datenregister
An	Adreßregister
Dn	Datenregister
Source	Quelloperand
Dest	Zieloperand
<ea>	effektive Adresse
#n	Integer-Wert
<list>	Registerliste (Zum Beispiel D0/D2-D4/A0-A3. Dies würde sich auf die Register D0 D2 D3 D4 A0 A1 A2 A3 beziehen, verwendet wird diese Bezeichnung der Register beim Befehl »MOVEM«.)

Wo immer Sie den Operanden »<ea>« sehen, können Sie alle normalen Adressierungsarten nutzen. Im einzelnen sind das folgende:

Adressierungsart	Assembler-Syntax
Datenregister direkt	Dn
Adreßregister direkt	An
Register indirekt	(An)
Register indirekt mit Postinkrementierung	(An)+
Register indirekt mit Prädekrementierung	-(An)
Register indirekt mit Offset	d16(An)
Register indirekt mit Index	d8(An,Rn)
Absolut Kurz	xxxx.W
Absolut Lang	xxxxxxxx
PC-relativ mit Offset	d16(PC)
PC-relativ mit Index	d8(PC,Rn)
Unmittelbar	#xxxx

In einigen Instruktionen (»Bcc«, »DBcc«, »Scc«) steht das Kürzel »cc« für den Bedingungscode. Die möglichen Bedingungscodes:

cc	Bedeutung	Bits
T	True (wahr)	
F	False (falsch)	
HI	HIgher (größer)	C'* Z'
LS	Lower or Same (kleiner oder gleich)	C + Z
CC,HS	Carry Clear, Higher or Same (größer gleich)	C'
CS,LO	Carry Set, Lower than (kleiner als)	C
NE	Not Equal (ungleich)	Z'
EQ	Equal (gleich)	Z
VC	oVerflow Clear (kein Überlauf)	V'
VS	oVerflow Set (Überlauf)	V
PL	PLus (positiv)	N'
MI	MInus (negativ)	N
GE	Greater than or Equal (größer oder gleich)	N*V+N'*V'
LT	Less Than (kleiner als)	N*V'+N'*V
GT	Greater Than (größer als)	N*V*Z'+N'*V'*Z'
LE	Less than or Equal (kleiner oder gleich)	Z + N*V'+N'*V

»*« bedeutet hier logisches UND, »+« bedeutet logisches ODER, und »'« steht für logisches NICHT.

Die folgende Liste führt alle Befehle auf, die der MC68000 kennt. Für weitere Informationen sollten Sie auf die Original-Dokumentation von Motorola zurückgreifen.

Mnemonic	Description
ABCD	Add binary coded decimal with extend
ADD	Add
ADDQ	Add quick
ADDX	Add with extend
AND	Logical AND
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Allways
BSET	Bit Test and Set
BSR	Branch To Subroutine
BSET	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
CMPM	Compare memory
DBcc	Test Condition, Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive OR
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move
MOVEM	Move Multiple Registers

Mnemonic	Description
MOVEP	Move Peripheral Data
MOVEQ	Move Quick
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Binary Coded Decimal with extend
NEG	Negate
NEGX	Negate with extend
NOP	No Operation
NOT	One's Complement
OR	Logical OR
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left Without Extend
ROR	Rotate Right Without Extend
ROXL	Rotate Left With Extend
ROXR	Rotate Right With Extend
RTE	Return From Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD	Subtract binary coded decimal with extend
Scc	Set Conditional
STOP	Stop
SUB	Subtract
SUBQ	Subtract quick
SUBX	Subtract with extend
SWAP	Swap data register halves
TAS	Test and Set Operation
TRAP	Trap
TRAPV	Trap on overflow
TST	Test
UNLK	Unlink

Diese Befehle sind nur die 56 Grundbefehle des 68000.

In dieser Liste sind bereits einige Kommandos in mehreren Variationen aufgeführt, zum Beispiel »ADD«, »ADDQ« und »ADDX«. In Wirklichkeit gibt es noch mehr Variationen, die von ASM-ONE automatisch umgesetzt werden. Beispielsweise behandelt ASM-ONE die folgenden Befehle gleich und verwendet den jeweils richtigen:

ADD	**normales Addieren**
ADDA	**Addiere zu einem Adreßregister**
ADDI	**Addiere unmittelbar**

Alle diese Befehle werden von Ihnen einfach wie ein normales »ADD« behandelt und erlauben dennoch alle Verwendungsmöglichkeiten der Instruktionen »ADDA« und »ADDI«. Dies ist bequemer für den Programmierer und bringt keine Probleme mit sich, da ASM-ONE anhand der Parameter eindeutig erkennen kann, welcher Befehl gemeint ist. Wenn Sie also beispielsweise:

```
ADDA.W   A0,A1
```

verwenden möchten, können Sie ebensogut schreiben:

```
ADD.W    A0,A1
```

Die einzige Einschränkung, die Sie sich merken sollten: Sie können Adreßregister nur wort- und langwortweise verwenden. Und das gilt natürlich auch dann noch, wenn Sie statt »ADDA« nun nur noch »ADD« schreiben. Hinzu kommt noch, daß bei einer wortweisen Addition zu einem Adreßregister das Wort zuvor auf ein Langwort erweitert wird, folgende Instruktionen also gleichwertig sind:

```
ADD.W    #1000,A0
ADD.L    #1000,A0 0
```

Die obere Instruktion braucht allerdings weniger Rechenzeit.

10.1.1 Befehle zum Kopieren von Daten

Die folgenden Instruktionen finden Anwendung, um Daten zwischen Speicher, Daten- und Adreßregistern zu bewegen.

Der grundlegende Befehl in dieser Kategorie ist der Befehl »**MOVE**«. Sie können mit »MOVE« Bytes, Wörter und Langwörter aus dem Speicher in Register übertragen, aus Registern in den Speicher schreiben, innerhalb des Speichers an andere Stellen kopieren oder innerhalb der Register kopieren.

Einige spezielle Varianten des Befehls »MOVE« erlauben Ihnen, das Bedingungscoderegister »CCR« zu beschreiben (»move <ea>,CCR«) und das Statusregister »SR« zu lesen (»move SR,<ea>«).

WARNUNG: Das Auslesen des Statusregisters ist auf höheren Prozessoren (68010, 68020 und Nachfolger) ein privilegierter Befehl, der im Supervisor-Modus abgearbeitet werden muß und andernfalls zu einer Privilege-Violation-Exception mit entsprechendem Guru führt.

Befehl	Syntax		Größe
EXG	EXG	Rx,Rx	L
LEA	LEA	<ea>,An	L
MOVE	MOVE	<ea>,<ea>	B W L
	MOVE	<ea>,CCR	W
	MOVE	<ea>,SR	W
	MOVE	SR,<ea>	W
	MOVE	USP,An	L
	MOVE	An,USP	L
MOVEM	MOVEM	<list>,<ea>	W L
MOVEM	MOVEM	<ea>,<list>	W L

10.1.2 Befehle für Integer-Arithmetik

Der MC68000 kann zwei Operanden addieren, subtrahieren, multiplizieren, dividieren und vergleichen. Einen einzelnen Operanden können Sie löschen, auf null testen, vorzeichenrichtig erweitern und negieren. Einige dieser Befehle brauchen allerdings eine Erklärung.

Wenn Sie zwei 64-Bit-Zahlen addieren möchten, ist der Befehl »ADDX« praktisch, da Sie wie folgt vorgehen können: Angenommen, die erste Zahl befindet sich in »D0« und »D1« und die zweite Zahl in »D2« und »D3«. Das Ergebnis möchten wir in »D2« und »D3« haben, das jeweils geringerwertige Langwort befinde sich in »D1« und »D3«. Die Addition läßt sich mit nur zwei Befehlen ausführen:

```
ADD.L   D1,D3
ADDX.L  D0,D2
```

Wenn das Carry-Flag bei der ersten Addition gesetzt wurde, wird es bei der zweiten ebenfalls hinzugefügt.

Auch die Divisionsbefehle »DIVS« und »DIVU« verdienen eine kurze Erläuterung. Hiermit können Sie einen 32-Bit-Wert durch einen 16-Bit-Wert dividieren. Das Ergebnis wird danach in den unteren 16 Bit des Zieloperanden abgelegt. In den oberen 16 Bit finden Sie danach den Rest der Ganzzahldivision.

Wenn Sie also die Funktion »MOD« der Sprache Pascal nachbilden wollen, können Sie das in wenigen Assembler-Befehlen tun. In einem Pascal-Programm würde »MOD« ungefähr wie folgt angewendet werden:

```
Result  :=  10 MOD 7
```

Natürlich lautet die Antwort »3«, doch der Computer muß es auch berechnen können:

```
START   MOVEQ   #10,D0
        MOVEQ   #7,D1
        DIVU    D1,D0
        CLR.W   D0
        SWAP    D0
        RTS     ; Rest 3 als Langwort in D0
```

Der Unterschied zwischen »DIVU« und »DIVS« ist, daß »DIVS« unter Berücksichtigung des Vorzeichens dividiert, während »DIVU« für vorzeichenlose Zahlen gedacht ist.

Befehl	Syntax		Größe
ADD	ADD	<ea>,Dn	B W L
	ADD	Dn,<ea>	B W L
ADDA	ADDA	<ea>,An	W L
ADDI	ADDI	#d,<ea>	B W L
ADDQ	ADDQ	#d,<ea>	B W L
ADDX	ADDX	Dy,Dx	B W L
	ADDX	-(Ay),-(Ax)	B W L
CLR	CLR	<ea>	B W L
CMP	CMP	<ea>,Dn	B W L
CMPA	CMPA	<ea>,An	W L
CMPI	CMPI	#d,<ea>	B W L
CMPM	CMPM	(Ay)+,(Ax)+	B W L
DIVS	DIVS	<ea>,Dn	W
DIVU	DIVU	<ea>,Dn	W
EXT	EXT	Dn	W L
MULS	MULS	<ea>,Dn	W
MULU	MULU	<ea>,Dn	W
NEG	NEG	<ea>	B W L
NEGX	NEGX	<ea>	B W L
SUB	SUB	<ea>,Dn	B W L
	SUB	Dn,<ea>	B W L
SUBA	SUBA	<ea>,An	W L
SUBI	SUBI	#d,<ea>	B W L
SUBQ	SUBQ	#d,<ea>	B W L
SUBX	SUBX	Dy,Dx	B W L
	SUBX	-(Ay),-(Ax)	B W L
TAS	TAS	<ea>	B
TST	TST	<ea>	B W L

10.1.3 Logische Operationen

Allen logischen Operationen ist gemeinsam, daß nur Datenregister und keine Adreßregister als Operanden verwendet werden dürfen. Auch sind keine zwei Speicheradressen als Operanden zulässig, dennoch sind die logischen Befehle des 68000 sehr nützlich. Erinnern wir uns nochmals an das Beispiel »MOD« aus dem vorhergehenden Unterkapitel:

```
        Result  = 10 MOD 7 = 3
```

Wenn die Frage

```
        Result  = 10 MOD 8 = ?
```

gewesen wäre, hätten wir das Problem auch mit einem logischen Befehl wie folgt lösen können:

```
START   MOVEQ    #10,D0      ; Zahl 1
        MOVEQ    #8,D1       ; Zahl 2
        SUBQ.L   #1,D1       ; minus 1
        AND.L    D1,D0       ; Bits ausmaskieren
        RTS
```

Diese Methode ist anwendbar, solange die zweite Zahl eine Zweierpotenz ist, also 2, 4, 8, 16, 32 etc., und funktioniert, da sich diese Werte mit 1, 2, 3, 4, 5 etc. Bits darstellen lassen, so daß die darüberliegenden Bits einfach ausmaskiert werden können, um den Rest herauszufinden. Da nur eine einzige logische Operation statt einer zeitaufwendigen Division durchgeführt werden muß, ist dieser Weg deutlich schneller.

Befehl	Syntax		Größe
AND	AND	<ea>,Dn	B W L
	AND	Dn,<ea>	B W L
ANDI	ANDI	#d,<ea>	B W L
	ANDI	#d,SR	B W
EOR	EOR	Dn,<ea>	B W L
EORI	EORI	#d,<ea>	B W L
	EORI	#d,SR	B W
NOT	NOT	<ea>	B W L
OR	OR	<ea>,Dn	B W L
	OR	Dn,<ea>	B W L
ORI	ORI	#d,<ea>	B W L
	ORI	#d,SR	B W

10.1.4 Schiebe- und Rotationsbefehle

Auch diese Befehlsgruppe ist sehr wichtig. Mit ihr können Sie 32-Bit-Zahlen in ihre einzelnen Bits zerlegen und dann weiterverarbeiten. Weiterhin können Sie mit ihr sehr schnell durch Zweierpotenzen dividieren, da bei jedem Shift durch zwei geteilt wird. Sie könnten also die folgenden Divisionsbefehle durch die schnelleren Schiebebefehle ersetzen:

```
DIVU   #8,D0   =   LSR.L   #3,D0
DIVS   #8,D0   =   ASR.L   #3,D0
```

Hier haben Sie zusätzlich den Vorteil, daß »LSR« und »ASR« auch mit Langwörtern arbeiten, während ein Ergebnis von »DIVU« oder »DIVS« nur 16 Bit anzubieten hätte. Die Schiebebefehle schieben nur alle Bits in eine Richtung, wobei das herausgeschobene Bit ins C- und X-Flag kopiert wird:

```
    abcdefgh
=   nabcdefg     C = h X = h
```

Bei »ASL«, »LSL« und »LSR« wird ein Null-Bit nachgeschoben, bei »ASR« hängt das eingeschobene Bit (hier: n) vom vormals höchstwertigen Bit ab.

Die Rotationsbefehle arbeiten vergleichbar den Schiebebefehlen, allerdings wird das herausgeschobene Bit am anderen Ende wieder eingefügt. Wenn Sie also ein Byte um acht Bits nach links verschieben, haben Sie erneut dasselbe Byte.

```
      abcdefgh
  =   habcdefg       C = h
```

Die Befehle »ROXL« und »ROXR« handhaben den Vorgang geringfügig anders. Hier wird das herausgeschobene Bit ins X-Flag geschoben, und das vormals dort befindliche Bit wird hereingeschoben. Hier müßten Sie also ein Byte neunmal schieben, um dasselbe Byte wieder zu erhalten.

```
      abcdefgh
  =   Xabcdefg       C = h X = h
```

Nach der nächsten Rotation:

```
      Xabcdefg
  =   hXabcdef       C = g X = g
```

Es ist mit den Rotationsbefehlen möglich, eine 32-Bit-Division zu programmieren:

Angenommen wird, daß Sie die beiden Operanden in »D0« und »D1« übergeben, zurückgegeben wird das Ergebnis der Division von »D1« durch »D0« in »D1«.

```
DIVIDE32:
        MOVEQ   #32,D3
        MOVEQ   #0,D2
.LOOP   SUB.L   D0,D2
        BCC.B   .OK
        ADD.L   D0,D2
```

```
.OK     ROXL.L   #1,D1
        ROXL.L   #1,D2

        DBF      D3,.LOOP
        NOT.L    D1
        RTS
```

Sie sehen, daß sich eine solche Routine bereits in wenigen Zeilen programmieren läßt.

Befehl	Syntax	Größe
ASL	ASL Dx,Dy	B W L
	ASL #d,Dn	B W L
	ASL <ea>	W
ASR	ASR Dx,Dy	B W L
	ASR #d,Dn	B W L
	ASR <ea>	W
LSL	LSL Dx,Dy	B W L
	LSL #d,Dn	B W L
	LSL <ea>	W
LSR	LSR Dx,Dy	B W L
	LSR #d,Dn	B W L
	LSR <ea>	W
ROL	ROL Dx,Dy	B W L
	ROL #d,Dn	B W L
	ROL <ea>	W
ROR	ROR Dx,Dy	B W L
	ROR #d,Dn	B W L
	ROR <ea>	W
ROXL	ROXL Dx,Dy	B W L
	ROXL #d,Dn	B W L
	ROXL <ea>	W
ROXR	ROXR Dx,Dy	B W L
	ROXR #d,Dn	B W L
	ROXR <ea>	W

Wenn Sie ein Register allerdings nur um ein oder zwei Bits nach links schieben wollen, ist es schneller, eine Addition durchzuführen.

Wollen Sie beispielsweise das Wort in »D0« verdoppeln, braucht

```
LSL.W    #1,D0
```

acht Taktzyklen,

```
ADD.W    D0,D0
```

aber nur vier Taktzyklen Wollen Sie ein Wort in »D0« vervierfachen, braucht

```
LSL.W    #2,D0
```

zehn Taktzyklen,

```
ADD.W    D0,D0
ADD.W    D0,D0
```

aber nur acht Taktzyklen

Wenn Sie aber ein Langwort in »D0« haben, ist die Addition nur noch beim Verdoppeln schneller als das Schieben.

10.1.5 Manipulationen auf Bit-Ebene

Die Befehle zur Bit-Manipulation werden verwendet, um ein bestimmtes Bit in einer Speicherzelle oder einem Datenregister zu setzen, zu löschen oder zu invertieren.

Beispielsweise können Sie in Ihren Programmen »D7« als globales Statusregister verwenden, um verschiedene Zustände zu speichern.

Beispiel:

```
CursorAn = 0
        BSET    #CursorAn,D7      ; Der Cursor ist
                                  ; eingeschaltet
```

Sehr praktisch ist die Tatsache, daß die Bit-Befehle die Flags entsprechend dem vorherigen Status des geänderten Bits setzen. Sie könnten also wie folgt abfragen:

```
        BCLR    #CursorOn,D7      ; Cursor löschen
        BNE     TheCursorWasSet
```

Befehl	Syntax	Größe
BTST	BTST Dn,<ea>	B L
	BTST #d,<ea>	B L
BSET	BSET Dn,<ea>	B L
	BSET #d,<ea>	B L
BCLR	BCLR Dn,<ea>	B L
	BCLR #d,<ea>	B L
BCHG	BCHG Dn,<ea>	B L
	BCHG #d,<ea>	B L

Langwortweise dürfen Sie diese Befehle nur in bezug auf Datenregister verwenden.

10.1.6 Befehle zur BCD-Arithmetik

Das BCD-Format (Binary Coded Decimal) ist ein spezielles Format, das Rundungsfehler bei der Umwandlung vermeiden hilft und es darüber hinaus erleichtert, Zahlen im Speicher in ASCII-Form auf dem Bildschirm auszugeben.

Normalerweise kann ein Byte die Werte von 0 bis 255 annehmen, doch in der binär codierten Dezimaldarstellung hat es nur noch den Wertebereich 0 bis 99. Wenn wir uns die zwei Nibbles (à vier Bits) in einem Byte ansehen, stellen wir fest, daß nur die ersten zehn der 16 möglichen den Dezimalzahlen entsprechen, denn nur diese werden in der BCD-Darstellung verwendet.

Binär	BCD-Darstellung
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
0000 0101	5
0000 0110	6
0000 0111	7
0000 1000	8
0000 1001	9
0001 0000	10
0001 0001	11

Wenn Sie also eine binär codierte Dezimalzahl in einen ASCII-String zur Bildschirmausgabe wandeln wollen, können Sie wie folgt verfahren:

```
BCDToStr:
         MOVEQ    #0,D0       ; D0 löschen
         MOVE.B   BinByte,D0    ; Byte in D0
                              ; einlesen D0 = $00xy
         LSL.W    #4,D0       ; 4 Bits links
                              ; schieben D0 = $0xy0
         LSR.B    #4,D0       ; 4 Bits rechts
                              ; schieben D0 = $0x0y
         ADD.W    #'00',D0    ; Addiere ASCII '00'
                              ; D0 = $3x3y
         MOVE.W   D0,String   ; In Puffer ablegen
         RTS
BinByte  DC.B     4<<4+6      ; Das Byte, von dem
                              ; wir ausgehen
         EVEN
String   DC.W     0           ; Hier wird der
                              ; String eingetragen
```

Wenn Sie zwei achtstellige BCD-Zahlen im Speicher haben und diese addieren wollen, können Sie wie folgt vorgehen:

```
AddBCD:
        LEA.L    BCD1+3(PC),A0  ; Letztes Byte
                              ; der ersten Zahl
        LEA.L    BCD2+3(PC),A1  ; Letztes Byte
                              ; der zweiten Zahl
        MOVE.W   #4,CCR       ; X-Flag löschen
        ABCD     -(A0),-(A1)
        ABCD     -(A0),-(A1)
        ABCD     -(A0),-(A1)
        ABCD     -(A0),-(A1)
        RTS
BCD1    DC.L     $12345678      ; erste BCD-Zahl
BCD2    DC.L     $87654321      ; zweite BCD-Zahl
```

Befehl	Syntax	Größe
ABCD	ABCD Dx,Dy	B
	ABCD -(Ay),-(Ax)	B
SBCD	SBCD Dx,Dy	B
	SBCD -(Ay),-(Ax)	B
NBCD	NBCD <ea>	B

10.1.7 Programm-Kontrollbefehle

Die Kontrollbefehle sind sehr hilfreich, und einige von ihnen haben wir auch schon in den Beispielen verwendet. Die neben dem Befehl »MOVE« wohl am meisten gebrauchte Instruktion in Assembler-Programmen ist die bedingte Verzweigung. Dies sind alle die Befehle, die in der Befehlsliste mit »cc« enden. Anstelle dieses »cc« setzen Sie bei der Anwendung des Befehls eine der logischen Bedingungen ein, die am Anfang dieses Kapitels bereits erläutert wurden, also zum Beispiel »EQ« oder »NE«. Sie verwenden die Kontrollbefehle folgendermaßen:

```
SUBQ.W   #1,D0
BNE      NotFinished    ;weiter bis D0 = 0

CMP.W    D0,D1
BHI      Higher       ; D1 größer als D0 ?
```

Bedingte Befehle:

Befehl	Syntax	Größe
Bcc	Bcc <Label>	B W (S L)
DBcc	DBcc Dn,<Label>	W
Scc	Scc <ea>	B

Unbedingte Befehle (Verzweigungen):

Befehl	Syntax	Größe
BRA	BRA <Label>	B W (S L)
BSR	BSR <Label>	B W (S L)
JMP	JMP < ea>	L
JSR	JSR <ea>	L

Rücksprungbefehle:

Befehl	Syntax	Aktion
RTR	RTR MOVE.W (SP)+,CCR	MOVE.L (SP)+,PC
RTS	RTS MOVE.L (SP)+,PC	

10.1.8 LINK und UNLINK

Diese Instruktionen werden in erster Linie von Hochsprachen-Compilern benutzt, beispielsweise in Compilern für C oder Pascal. Sie werden normalerweise verwendet, um für Subroutinen Platz auf dem Stapel zu reservieren.

Natürlich gibt es auch für den Assembler-Programmierer Situationen, in denen er diese Befehle gut gebrauchen kann, beispielsweise wenn er rekursive Funktionen schreibt und nicht bei jedem Durchlauf extra Speicher reservieren möchte.

Befehl	Syntax	Vorgang
LINK	LINK An,#d	MOVE.L An,-(SP)
		MOVEA.L SP,An
		ADD.W #d,SP
UNLK	UNLK An	MOVE.L An,SP
		MOVE.L (sp)+,An

10.1.9 System-Kontrollbefehle

Unter den Befehlen, die in diese Kategorie fallen, ist der Befehl »STOP« derjenige, der am ehesten Erklärung braucht.

Der Befehl »STOP« ist privilegiert, da er schreibend auf das Statusregister zugreift. Nach der Ausführung des Befehls »STOP« befindet sich der Prozessor in einen Wartezustand, aus dem er nur durch einen Interrupt herausgeholt werden kann. Diese Instruktion ist nützlich, um wertvolle Buszyklen zu sparen.

Sehr wichtig ist ebenfalls, daß der Befehl »MOVE SR,<ea>« nur auf dem 68000er im User-Modus aufgerufen werden kann, während alle anderen Prozessoren der MC68000-Reihe sich zu seiner Ausführung im Supervisor-Modus befinden müssen.

Privilegierte Befehle:

Befehl	Syntax	Aktion
RESET	RESET	
RTE	RTE	MOVE.W (SP)+,SR MOVE.L (SP)+,PC
STOP	STOP #d	MOVE.W #d,SR und Stop
ANDI	ANDI #d,SR	
EORI	EORI #d,SR	
ORI	ORI #d,SR	
MOVE	MOVE <ea>,SR	
	MOVE USP,An	
	MOVE An,USP	

Trap-erzeugende Instruktionen:

Befehl	Syntax	Aktion
TRAP	TRAP #<vector>	
TRAPV	TRAPV	Wenn V = 1 dann Trap
CHK	CHK <ea>,Dn	

Statusregister-Instruktionen:

Befehl	Syntax	Aktion
ANDI	ANDI.B #d,CCR	
EORI	EORI.B #d,CCR	
ORI	ORI.B #d,CCR	
MOVE	MOVE <ea>,SR	
	MOVE SR,<ea>	Privilegiert auf MC68010 und höher

10.2 Exceptions oder Ausnahmeroutinen

In einem Programm führt man normalerweise eine bestimmte Unterroutine aus, wenn eine spezielle Funktion benötigt wird. Wenn dem Prozessor spezielle Umstände zustoßen, führt er automatisch eigene Unterroutinen aus, die »Exceptions« oder »Ausnahmebehandlungen« genannt werden.

Exceptions unterscheiden sich von normalen Unterprogrammen nur dadurch, daß sie nicht durch einen »RTS«-Befehl, sondern durch den Befehl »RTE« (Return from Exception) abgeschlossen werden müssen. Die Ausnahmeroutinen des 68000 sind folgende:

Vektor	Adresse	Zuordnung
0	$000	RESET: Initialwert SSP
-	$004	RESET: Initialwert PC
2	$008	Busfehler
3	$00c	Adreßfehler
4	$010	illegale Instruktion
5	$014	Division durch null
6	$018	CHK-Vektor
7	$01c	TRAPV-Vektor
8	$020	Privilegverletzung
9	$024	Trace
10	$028	LINE_A-Emulation
11	$02c	LINE_F-Emulation
12 - 14	$030 - $03b	reserviert
15	$03c	nicht initialisierter Interrupt
16 - 23	$040 - $05f	reserviert
24	$060	Spurious Interrupt
25	$064	Level 1 Interrupt Autovektor
26	$068	Level 2 Interrupt Autovektor
27	$06c	Level 3 Interrupt Autovektor
28	$070	Level 4 Interrupt Autovektor
29	$074	Level 5 Interrupt Autovektor
30	$078	Level 6 Interrupt Autovektor
31	$07c	Level 7 Interrupt Autovektor
32 - 47	$080 - $0bf	TRAP-Instruktionsvektoren
48 - 63	$0c0 - $0ff	reserviert
64 - 255	$100 - $3ff	Benutzer-Interrupts

RESET: Initialwert SSP

Von hier wird der Initialwert für den Supervisor-Stackpointer beim Reset des Systems geholt. Sie werden allerdings an dieser Adresse nicht den »SSP« finden, da hier nur beim Start des Systems das Amiga-ROM eingeblendet wird. Danach ist die Adresse »0« unbenutzt.

RESET: Initialwert PC

Von hier wird der Initialwert für den Programmzähler beim Reset des Systems geholt. Sie werden allerdings an dieser Adresse nicht den PC finden, da hier nur beim Start des Systems das Amiga-ROM eingeblendet wird. Danach enthält Adresse »4« die Adresse der »Exec-Base«.

Busfehler

Diese Exception wird aufgerufen, wenn auf reservierten oder nicht zugänglichen Speicher zugegriffen wurde.

Adreßfehler

Tritt auf, wenn auf eine ungerade Adresse zugegriffen wird.

Illegale Instruktion

Eine illegale Instruktion ist ein 16-Bit-Binärmuster, dem keiner der gültigen MC68000-Befehle zugeordnet ist. Dazu gehört auch der Opcode »$4afc«, der in der Assembler-Sprache direkt als »ILLEGAL« verwendet werden kann, um diese Exception gezielt auszulösen.

Division durch null

Tritt auf, wenn durch null geteilt werden soll; zum Beispiel »DIVS #0,D0«.

CHK-Vektor

Ist reserviert für die Instruktion »CHK«. Über diesen Vektor kann eine Fehlerbehandlung gestartet werden, wenn eine »CHK«-Bereichsüberprüfung ergibt, daß der zulässige Wertebereich überschritten ist.

TRAPV-Vektor

Ist reserviert für die »TRAPV«-Instruktion. Die Instruktion »TRAPV« löst diese Exception aus, wenn das V-(Overflow-)Bit im Register »CCR« gesetzt ist.

Privilegverletzung

Wird ausgelöst, wenn Sie einen privilegierten Befehl im User-Modus verwenden, zum Beispiel »RESET« oder »MOVE.W xx,SR«.

Trace

Wird bei gesetztem Trace-Bit im Supervisor-Register nach jedem Befehl aufgerufen, um eine Abarbeitung in Einzelschritten zu ermöglichen.

LINE_A-Emulation

Auf dem MC68000 sind nicht alle möglichen Opcodes verwendet worden, da einige für spätere Verwendung reserviert wurden. Dies sind in erster Linie die Befehls-Opcodes, die mit »$A« oder »$F« beginnen. Alle Opcodes, die mit »$A« beginnen, werden »LINE_A«-Befehle genannt. Sie finden Verwendung bei der Implementation eigener Befehle oder bei der Ansteuerung von Coprozessoren.

LINE_F-Emulation

Entspricht der »LINE_A«-Emulation, lediglich für die Befehle, deren Opcode mit »$F« beginnt.

11. Programmierbeispiele

In diesem Kapitel werden wir verschiedene Programmiertechniken und Möglichkeiten zur Lösung von Programmierproblemen kennenlernen.

Auf der Programmdiskette befinden sich folgende Beispielprogramme im Verzeichnis »**Examples**«.

GettingStarted.S

Dies ist ein Quelltext, der auf der System-Startup-Routine basiert. Er zeigt einfach nur ein fraktales Bild an.

SystemStartUp.S

Wenn Sie ein Programm unter Verwendung der Betriebssystemroutinen schreiben (wie zum Beispiel ASM-ONE), dann gibt es verschiedene Grundfunktionen, die Sie dabei immer brauchen. Die meisten davon finden Sie in diesem Startup-Modul.

BinaryConv.S

Konvertiert einen Registerinhalt in einen binären ASCII-String. Dieses Beispiel ist für die ersten Schritte mit dem Debugger gedacht.

BootBlock.S

Erzeugt einen eigenen Bootblock, den Sie mit einem eigenen Text versehen können.

NonSystemStartUp.S

Nützlicher Startup-Code für alle Hardware-nahen Programme und Routinen.

ScrollExample.S

Eine kleine Scrolltext-Routine, die auf der Non-System-Startup-Routine basiert.

LineDraw.S

Basiert wie »ScrollExample« auf der Non-System-Startup-Routine.

WindowExample.S

Ein Beispiel für die Verwendung von Includefiles. Öffnet ein Fenster, liest einen String von der Tastatur ein und gibt ihn im Fenster aus.

Directory.S

Basiert auf Includes wie das Window-Beispiel. Öffnet ein Fenster und gibt darin ein Directory aus.

Die obigen Beispiele sind weitestgehend selbsterklärend und wurden auf die Diskette aufgenommen, weil sie zu lang zum Abdrucken sind. Sie können Teile dieser Quellcodes auch in Ihren eigenen Programmen verwenden, aber ich rate Ihnen, daß Sie nur versuchen, die Funktionsweise der Beispiele zu verstehen und dieses Wissen in Ihren Programmen zu verwenden.

Diesem Zweck dienen auch die Beispielprogramme, die in diesem Handbuch abgedruckt sind. Es ist natürlich möglich, viele davon zu optimieren, aber ich fand es sinnvoller, verschiedene Programme für mehrere Probleme vorzustellen als nur eine Handvoll hochoptimierter Algorithmen.

Die nun folgenden Programmierbeispiele lassen sich in drei Kategorien aufteilen:

Allgemeine Beispiele: Programme, die auf jedem MC68000-System lauffähig sind

Systemprogrammierung: Programme, die auf das Amiga-Betriebssystem zurückgreifen

Hardware-Programmierung: Programme, die die Amiga-Hardware direkt nutzen

11.1 Allgemeine Beispiele

In den folgenden Beispielen werden wir verschiedene Methoden zur Lösung einiger einfacher Probleme diskutieren. Die Ausführungszeit der einzelnen Routinen wird angegeben, um zu vergleichen, welche Problemlösung die schnellste ist.

Wenn Sie ein absoluter Anfänger im Bereich Maschinensprache sind, sollten Sie dieses Unterkapitel überspringen und es erst durcharbeiten, wenn Sie sich einige Grundkenntnisse angeeignet haben.

11.1.1 Langwort-Multiplikation

Der MC68000 kann zwar 16-Bit-Werte mit einem einzigen Befehl multiplizieren, aber in gewissen Anwendungen ist es erforderlich, 32-Bit-Zahlen miteinander zu multiplizieren. Zunächst ein wenig Theorie:

Wir betrachten zwei zweistellige Zahlen, wobei wir jede Ziffer durch einen anderen Buchstaben repräsentieren:

```
xy * mn
```

Wenn wir diese beiden Zahlen miteinander multiplizieren wollen, aber nur noch wissen, wie man einstellige Zahlen multipliziert (weil wir das große Einmaleins schon wieder vergessen haben), gehen wir wie folgt vor:

```
  | x | y |              |  y*n  |
* | m | n |    =      |  x*n  |      * 10
                      |  y*m  |      * 10
                  |  x*m  |          * 100
                    ---------------
               = | a | b | c | d |
```

Wie man sieht, erhalten wir eine vierstellige Zahl. Anstatt nun die einzelnen Ziffern zu betrachten, gehen wir zu 16-Bit-Zahlen über. Das nun folgende Programm multipliziert zwei 32-Bit-Zahlen mittels vier 16-Bit-Multiplikationen:

Wir haben:

	HiWord	LoWord
D0 =	x	y
D1 =	m	n

Das Resultat wird in »D0« (höherwertige 32 Bit) und »D1« (niederwertige 32 Bit) abgelegt:

```
MULU32  MOVEM.L  D2/D3/D4,-(A7)    ; Sichere
                                 ; verwendete Register
        MOVE.W   D1,D2           ; d2 = n
        MOVE.L   D1,D3           ; d1 = n
        SWAP     D3              ; d3 = m
        MOVE.W   D3,D4           ; d4 = m

        MULU     D0,D1           ; d1 = y*n
        MULU     D0,D3           ; d3 = y*m
        SWAP     D0
        MULU     D0,D2           ; d2 = x*n
        MULU     D4,D0           ; d0 = x*m
```

```
SWAP D1
ADD.W       D2,D1       ; d1 = d1 + d2<<16
CLR.W       D2
SWAP        D2
ADDX.L      D2,D0       ; d0 = d0 + d2>>16
                        ; + Carry
ADD.W       D3,D1
CLR.W       D3
SWAP        D3
ADDX.L      D3,D0       ; d0 = d0 + d2>>16
                        ; + Carry
SWAP        D1          ; D1 wieder in rich-
                        ; tige Reihenfolge

MOVEM.L     (A7)+,D2/D3/D4
RTS
```

Diese 32-Bit-Multiplikationsroutine braucht nur fünf Register. Genauso ist es auch möglich, zwei vorzeichenbehaftete 32-Bit-Werte zu multiplizieren, aber das überlasse ich Ihnen als Übung.

Ich habe spaßeshalber auch die Ausführungszeit für dieses Beispielprogramm ausgerechnet, die 432 Taktzyklen beträgt. Wenn Sie die Routine allerdings als ein Makro verwenden (also ohne das »RTS«) und erlauben, daß »D2«, »D3« und »D4« verändert werden, braucht die Routine nur noch maximal 348 Zyklen. (Dies ist der Maximalwert für den ungünstigsten Fall, da ein »MULU« maximal 70 Zyklen braucht.)

Natürlich ist es nach dieser Methode auch möglich, ein Programm zu schreiben, das Zahlen mit mehr als 32 Bit multipliziert, doch Werte mit dieser Genauigkeit werden nur sehr selten verwendet. Normalerweise greift man dann auf Fließkomma-Arithmetik zurück.

11.1.2 Langwort-Division

Wo wir gerade bei der 32-Bit-Integerarithmetik sind, möchte ich nun nochmals auf die Divisionsroutine zurückkommen, die bereits bei der Vorstellung des Befehlssatzes angesprochen wurde.

Hier ist eine optimierte Version, die in »D1« den Quotienten aus »D1« und »D0« zurückgibt:

```
DIVIDE32b:
      MOVEQ    #0,D2

      REPT     32
       SUB.L   D0,D2
       BCC.B   *+2          ; Springt zur aktu-
                            ; ellen Adresse +2
       ADD.L   D0,D2        ; wird gegebenfalls
                            ; übersprungen
       ADDX.L  D1,D1
       ADDX.L  D2,D2
      ENDR
      NOT.L    D1
      RTS
```

Diese Routine braucht maximal 1288 und mindestens 1096 Zyklen.

11.1.3 Löschen, Kopieren und Vergleichen

Nachdem wir diese einfachen mathematischen Routinen implementiert haben, gehen wir zu einem anderen wichtigen Gebiet über: das Löschen, Kopieren und Vergleichen von Daten. Zunächst das Löschen:

Bevor Sie eine Methode zum Löschen von Daten auswählen, sollten Sie einen Moment über den konkreten Fall nachdenken. Gibt es eine oder gar mehrere konstante Bedingungen?

Nehmen wir einmal an, daß Sie den Bildspeicher löschen wollen. Sie können hierzu natürlich auf den Blitter zurückgreifen - doch was ist, wenn der Blitter bereits mit etwas anderem beschäftigt ist? Der erste Versuch könnte eine Schleife sein:

```
CLEAR1  LEA.L   SCREEN,A0
        MOVE.W  #80*200-1,D0          ; Bytes - 1
.LOOP   CLR.B   (A0)+
        DBF     D0,.LOOP
```

Dies kann bereits beträchtlich verbessert werden, wenn Sie mit dem Befehl »MOVE.L D1,(A0)+« ein ganzes Langwort auf einmal löschen. »CLR.L (A0)+« sollten Sie nicht verwenden, da die »CLR«-Instruktion auf dem 68000er vor dem Schreibzugriff noch lesend zugreift.

Sie könnten die Instruktion auch viele Male hintereinander folgen lassen, so daß Sie die Zyklen für die »DBF«-Instruktion einsparen:

```
CLEAR2  MOVEQ    #0,D0
        LEA.L    SCREEN,A0
        REPT     80*200/4
         MOVE.L  D0,(A0)+
        ENDR
        RTS
```

Dies erlaubt Ihnen, 2387000 Byte pro Sekunde zu löschen - aber das Programm ist 12800 Byte lang.

Und es geht auch noch schneller: Wenn Sie das Kommando »MOVEM« benutzen und alle (vorher geleerten) Register mit einem Befehl in den Speicher schreiben, ist die vorher erreichte Geschwindigkeit noch zu überbieten.

```
CLEAR3  MOVEM.L   BLANK,D0-A1
        LEA.L     SCREENEND,A2
        REPT      2*200
         MOVEM.L  D0-A1,-(A2)
        ENDR
        RTS
```

Diese Routine löscht 3240000 Byte pro Sekunde und belegt »nur noch« 1600 Byte.

Dieses Beispiel beschäftigte sich mit einem Datenfeld, über das Sie alles wußten - insbesondere die Größe. Nehmen wir nun ein Beispiel, in dem Sie gar nichts wissen: Die Routine bekommt eine Position und die Länge übergeben und kein Byte mehr.

Auch hier könnten wir wieder die Routine »CLEAR1« verwenden, und auch unter diesen Umständen läßt sie sich noch beschleunigen. Der naheliegendste Schritt ist auch hier, die Löschanweisung mehrfach hintereinander zu schreiben:

(Die Routine wird mit einem Pointer in »A0« und der Länge in »D0« aufgerufen.)

```
CLEAR1.2
        MOVEQ    #0,D1       ; Füllwert = 0
        SUBQ.L   #8,D0       ; Dekrementiere
                             ; Zähler um 8
        BMI.B    .NOMORE     ; Keine ganzen 8
                             ; mehr?

.LOOP   REPT     8
        MOVE.B   D1,(A0)+    ; Lösche 8 Bytes
        ENDR
        SUBQ.L   #8,D0       ; noch 8 mehr
        BPL.B    .LOOP       ; fertig
```

```
.NOMORE ADDQ.L  #7,D0      ; wie viele noch?
        BMI.B   .FINI      ; keines?
.LOOP2  MOVE.B  D1,(A0)+   ; lösche die
                           ; restlichen mit
        DBF     D0,.LOOP2    ; einer DBF-
                           ; Schleife
.FINI   RTS
```

Diese Routine ist schon recht schnell, aber auch sie läßt sich noch beschleunigen, wenn die Adresse zunächst auf eine Wortgrenze justiert wird, so daß Sie die Langwort-Instruktionen verwenden können.

Ein Schema für ein solches Programm folgt unten. Das Programm ist noch nicht in Assembler umgesetzt, da dies eine gute Übung für Sie darstellt. Beachten Sie bitte, daß es besser ist, die Adresse auf ein Langwort statt nur auf ein Wort auszurichten, weil auf dem MC68020 und höher »MOVE.L« schneller ist, wenn es auf eine Langwortgrenze ausgerichtet ist. Dies liegt daran, daß diese Prozessoren einen 32-Bit-Datenbus anbieten, und ein Langwortzugriff auf eine nicht durch 4 teilbare Adresse dann trotzdem zwei Speicherzugriffszyklen braucht.

CLEAR1.3

Adresse auf Langwortgrenze anpassen, indem die ersten 0 bis 3 Bytes einzeln gelöscht werden
Langwortweise löschen
Letzte 0 bis 3 Bytes wieder einzeln löschen
RTS

Diese Routine ist um so schneller, je größer der zu löschende Bereich ist. Ab 10 bis 20 Byte (je nach Implementation) ist sie generell schneller als die erste Routine. In der optimalen Version ist diese Routine dreimal schneller als die erste und siebenmal schneller als »CLEAR1«.

Dieser Algorithmus ist auch verwendbar, um Speicherinhalte zu verschieben oder zu vergleichen.

Ich möchte Ihnen hier noch eine nützliche Kopierroutine zeigen, die nach dem Prinzip arbeitet, das in »CLEAR1.2« verwendet wurde.

Sie rufen diese Routine mit einem Zeiger auf die Quelladresse in »A0«, einem Zeiger auf die Zieladresse in »A1« und der Länge in »D0« auf:

```
COPY1.2
        SUBQ.L  #8,D0
        BMI.B   .NOMORE

.LOOP   REPT    8
        MOVE.B  (A0)+,(A1)+
        ENDR
        SUBQ.L  #8,D0
        BPL.B   .LOOP

.NOMORE ADDQ.L  #7,D0
        BMI.B   .FINI
.LOOP2  MOVE.B  (A0)+,(A1)+
        DBF     D0,.LOOP2
.FINI   RTS
```

11.1.4 Sortieren

Sortieren ist eine weitere wichtige Anwendung, und es gibt viele verschiedene Algorithmen, um eine Sortierung durchzuführen. Einer der einfachsten und bekanntesten ist der Bubblesort-Algorithmus. Er funktioniert wie folgt:

Wir betrachten die folgenden fünf Ziffern

3 4 2 5 1

Bubblesort betrachtet immer nur zwei Ziffern und tauscht sie, wenn die erste größer als die zweite ist. Nach dem ersten Paar wird das zweite bearbeitet, dann das dritte und so weiter. Wenn nach einem kompletten Durchlauf kein Tausch mehr ausgeführt werden konnte, sind die Nummern fertig sortiert.

3 2 4 1 5	getauscht: 4-2 5-1
2 3 1 4 5	getauscht: 3-2 4-1
2 1 3 4 5	getauscht: 3-1
1 2 3 4 5	getauscht: 2-1
1 2 3 4 5	getauscht: keine

Wie Sie sehen können, braucht der Algorithmus fünf Durchläufe, um fünf Zahlen zu sortieren, also 5*5=25 Vergleiche.

Man sagt, daß die Effizienz dieses Algorithmus O(n*n) ist. Das bedeutet, daß n Elemente (n*n)-mal miteinander verglichen werden müssen, um sie zu sortieren. Wenn dafür jedesmal beispielsweise 72 Zyklen benötigt werden (also 100000 Vergleichs- und Tauschoperationen in der Sekunde durchgeführt werden), dauert es zehn Sekunden, um 1000 Elemente zu sortieren.

Die Bubblesort-Routine kann allerdings effizienter implementiert werden. Zunächst fällt auf, daß Listen mit nur einer vertauschten Zahl je nach ihrer Position in völlig verschiedenen Zeiten sortiert werden:

5 1 2 3 4	sortiert in zwei Durchläufen
2 3 4 5 1	sortiert in fünf Durchläufen

Es genügt eine minimale Änderung, um dieses Manko zu beheben. Wenn wir beim ersten Durchlauf von links nach rechts, beim zweiten aber von rechts nach links rückwärts vergleichen, erreicht Bubblesort schon bessere Zeiten:

5 1 2 3 4	sortiert in zwei Durchläufen
2 3 4 5 1	sortiert in drei Durchläufen

Wenn wir diese Methode anwenden, erhalten wir für das erste Beispiel »3 4 2 5 1«:

3 2 4 1 5	getauscht: 4-2 5-1
1 3 2 4 5	getauscht: 4-1 2-1 3-1
1 2 3 4 5	getauscht: 3-2
1 2 3 4 5	getauscht: keine

Nun braucht die Routine nur noch vier Durchläufe statt vorher fünf.

Wenn wir noch weiter optimieren wollen, können wir folgendes tun: Man merkt sich bei einem Durchlauf, an welcher Position das erste Element getauscht werden mußte, und führt beim nächsten Durchlauf erst eine Position vor dieser den ersten Vergleich durch.

Wir könnten noch mehr Umstände prüfen, doch darf dabei nicht vergessen werden, daß die Routine, wenn sie sehr viele Tests durchführt, unter Umständen länger mit den Tests beschäftigt ist, als das Sortieren ohne alle Tests benötigt hätte.

Die Version des Bubblesort, die ich selbst üblicherweise benutze, ist folgende:

Überprüfe die beiden Elemente. Wenn sie getauscht werden müssen, tausche sie, und gehe ein Paar zurück. Wenn nicht getauscht wird, gehe ein Paar vorwärts. Die Reihe ist sortiert, wenn das Ende erreicht ist.

Die Resultate dieser Routine sind recht gut. Normalerweise arbeitet sie vier- bis fünfmal schneller als die erste Routine, die ich Ihnen vorgestellt habe.

In 68000-Assembler sieht die Routine wie folgt aus:

```
BUBBLE_A_LA_RUNE:
        LEA     ListStart,A0  ; A0 =Startzeiger
        LEA     ListEnd,A1    ; A1 = Endzeiger
        MOVEQ   #ElemLen,D1   ; D1 = Element-
                              ; Länge
        MOVE.L  A0,A2       ; A2 = Zeiger auf
                            ; aktuelles Element
        MOVE.L  A0,A3
        ADD.L   D1,A3       ; A3 = Zeiger auf
                            ; nächstes Element
.LOOP   MOVE.B  (A3),D0
        CMP.B   (A2),D0     ; Element A =< Ele
                            ; ment B?
        BLO.B   .CHANGE     ; Nicht tauschen
        ADD.L   D1,A2
        ADD.L   D1,A3
        CMP.L   A3,A1       ; Ende erreicht?
        BNE.B   .LOOP       ; Sonst weiter
        RTS

.CHANGE MOVE.L  D1,D2       ; Tausche Elemente
.LOOP2  MOVE.B  (A2),D0
        MOVE.B  (A3),(A2)+
        MOVE.B  D0,(A3)+
        SUBQ.L  #1,D2
        BNE.B   .LOOP2

        SUB.L   D1,A2
        SUB.L   D1,A3
        CMP.L   A2,A0       ; Am ersten angekom
                            ; men?
        BEQ.B   .LOOP       ; Ja, zurück zur
                            ; Schleife
        SUB.L   D1,A2       ; Ein Element vor
        SUB.L   D1,A3       ; Ein Element zurück
        BRA.B   .LOOP       ; Und nochmals
```

Diese Implementation ist sehr allgemein gehalten. Wenn Sie lediglich Elemente fester Länge sortieren wollen und diese Länge durch 2 teilbar ist, können Sie eine Swap-Routine mit Langwort-Instruktionen schreiben. Aber natürlich wird auch die obenstehende Routine zuverlässig arbeiten.

Bubblesort arbeitet am besten, wenn Sie nur kleine Datenfelder haben oder Datenfelder, die weitgehend vorsortiert sind.

11.2 Systemprogrammierung

Das Amiga-Betriebssystem bietet viele Möglichkeiten, die zu umfangreich sind, um sie hier alle detailliert zu beschreiben. Dieses Unterkapitel wird Ihnen aber anhand einiger Beispiele aufzeigen, was Sie mit dem Betriebssystem erreichen können. Weitere Details entnehmen Sie am besten den »ROM Kernal Manuals« von Commodore (vergleiche Literaturhinweise).

Das System des Amiga basiert auf verschiedenen Libraries. Die meisten davon befinden sich im ROM, aber einige auch im Verzeichnis »LIBS:« auf Diskette und müssen vor ihrer Verwendung geladen werden. Die wichtigsten sind:

exec.library

Dies ist die Basis-Library. Mit ihr können Sie andere Libraries öffnen, Speicher allozieren, Tasks erzeugen etc.

dos.library

Enthält die Grundfunktionen zur Ein- und Ausgabe, gleich ob auf Laufwerke, Drucker oder den Bildschirm.

intuition.library

Handhabt Screens, Windows, Menus etc.

graphics.library

Enthält Routinen zum Zeichnen, Füllen, Scrollen, und Löschen von Grafiken und Grafikelementen.

layers.library

Kümmert sich um überlappende Bildbereiche.

diskfont.library

Diese Library ist auf der Diskette und enthält Routinen, um andere Zeichensätze verwenden zu können.

icon.library

Funktionen zum Erzeugen und Verändern von Icons auf dem Workbench-Screen

translator.library

Wird zusammen mit dem Software-mäßigen Sprachsynthesizer (»narrator.device«) verwendet, übersetzt englischen Text in die entsprechenden phonetischen Symbole.

mathffp.library

Enthält die einfachen Fließkommaroutinen.

mathieeedoubbas.library

Rechenroutinen mit doppelter Genauigkeit.

mathtrans.library

Komplexere Fließkommafunktionen wie »SIN«, »COS«, »LOG«, »EXP« etc.

Es gibt noch weitere Libraries, die aufgeführten sind aber die gebräuchlichsten. Normalerweise verwendet man bei der Benutzung von Libraries die entsprechenden Include-Files, die Sie in einem eigenen Verzeichnis auf Ihrer ASM-ONE-Diskette vorfinden. Sie enthalten die Konstanten (zum Beispiel Offsets, Flag-Definitionen) und Makrodefinitionen zu den Libraries.

Bevor Sie allerdings ein Include-File nutzen können, müssen Sie ASM-ONE verraten, in welchem Pfad es sie suchen soll. Das erledigt folgendes Kommando:

```
INCDIR   df0:include/
```

Hier befinden sich normalerweise die Include-Files, vorausgesetzt Sie haben die ASM-ONE-Diskette im Laufwerk »DF0:«.

Binden Sie nun folgende Include-Files ein, die wir in den nächsten Beispielen brauchen werden:

```
include   exec/exec_lib.i
include   libraries/dos_lib.i
include   libraries/dos.i
include   intuition/intuition_lib.i
include   intuition/intuition.i
```

Um eine Library zu öffnen, verfahren Sie wie folgt - angenommen die »dos.library« soll geöffnet werden.

```
DOS_LIBOPEN
        LEA.L     _DOSNAME(PC),A1   ; A1 =>
                                    ; dos.library
        MOVEQ     #0,D0             ; Version 0
        CALLEXEC  OpenLibrary       ; Library öffnen
        MOVE.L    D0,_DOSBASE       ; _dosbase =>
                                    ; library
        BEQ.L     ERROR_ERROR       ; Nullpointer =
                                    ; Fehler
        RTS
```

```
_DOSNAME DOSNAME          ; Dosname ist ein
                          ; Makro, das den String
                          ; 'dos.library',0 enthält
_DOSBASE DC.L 0           ; Pointer auf die
                                  ; Library
```

Nach dem Aufruf dieser Routine wird ein Zeiger auf die Library zurückgegeben. Wenn die »dos.library« nicht geöffnet werden könnte, wird eine Null zurückgegeben. Allerdings ist dann etwas sehr Schwerwiegendes schiefgelaufen ...

Wenn Sie die Include-Files nicht verwenden, müssen Sie den Offset »OpenLibrary« wie folgt definieren

```
        OpenLibrary         =   -552
```

und die Routine sähe so aus

```
DOS_LIBOPEN_NO_INCLUDE:
         LEA.L    _DOSNAME(PC),A1    ; A1 =>
                             ; dos.library
         MOVEQ    #0,D0      ; Version 0
         MOVE.L   $4.W,A6    ; Zeiger auf
                             ; exec.library
         JSR      OpenLibrary(A6)    ; Library
                                     ; öffnen
         MOVE.L   D0,_DOSBASE        ; _dosbase=>
                                     ; library
         BEQ.L    ERROR_ERROR        ; Nullpoin-
                                     ; ter=Fehler
         RTS

_DOSNAME DC.B     'dos.library'      ; Name (in
                             ; Kleinschrift!)
_DOSBASE DC.L     0          ; Pointer auf die
                             ; Library
```

Der Vorteil bei Include-Files ist, daß Sie sich nicht mehr alle Library-Offsets merken müssen. Im obigen Beispiel war das der Offset »OpenLibrary« mit einem Wert von -552.

Die einzige Library, deren Adresse direkt bekannt ist, ist die »exec.library«. Der Zeiger auf die »exec.library« befindet sich im Speicher an Adresse »$4«. Exec erfüllt eine Vielzahl von Aufgaben, sämtliche Exec-Routinen finden Sie in der Datei »exec_lib.i«. Wenn Sie eine solche Exec-Routine aufrufen wollen, müssen Sie das an der Adresse »ExecBase+Offset« tun. Da alle Offsets negativ sind, folgt, daß die Einsprungadressen vor der Basis der Library liegen. Dort befinden sich tatsächlich nicht die Routinen, sondern nur Sprungbefehle, die in die Routine springen.

Alle Library-Offsets folgen in Sechserschritten aufeinander, da ein absoluter Sprung 6 Bytes belegt. Die ersten Einträge in der Exec-Offset-Tabelle sind folgende:

```
Supervisor  = -30
ExitIntr    = -36
Schedule    = -42
```

Und viel weiter unten in der Liste finden wir

```
OpenLibrary  = -552
```

Wenn wir an der Adresse »ExecBase-552« nachsehen, finden wir etwas wie dieses:

```
$0000044E  JMP  $00005E3A    ; Verlassen Sie
                  , sich nicht auf diesen Wert,
                  ; er könnte sich in zukünf-
                  ; tigen Betriebssystemversio-
                  ; nen ändern
```

Sie sehen, daß in Wirklichkeit nur ein Sprungbefehl aufgerufen wird, der die Routine startet.

In den restlichen Beispielen werden die Definitionen aus den Include-Files genutzt. Wollen Sie sehen , was exakt mit Ihrem Quelltext passiert, disassemblieren Sie die Routinen. Um alle Include-Definitionen zu sehen, assemblieren Sie mit »=S«.

Nachdem wir die Library geöffnet haben, brauchen wir auch eine Routine, die sie wieder schließt:

```
DOS_LIBCLOSE
        MOVE.L    _DOSBASE(PC),A1    ; Pointer
                                     ; auf die Library
        CALLEXEC  CloseLibrary ; Schließe sie!
        RTS
```

Nach dem Öffnen der »dos.library« können wir sehr einfach ein Ausgabefenster auf dem Workbench-Screen öffnen.

```
WINDOW_OPEN
        MOVE.L    #_WINDOWNAME,D1      ; Window-
                                       ; Definition
        MOVE.L    #MODE_OLDFILE,D2     ; CON: muß
                                       ; existieren
        CALLDOS   Open
        MOVE.L    D0,_WINDOWHANDLE     ; Das Window-
                                       ; Handle
        BEQ.L     ERROR        ; Null, wenn Fenster
                               ; nicht geöffnet wer-
                               ; den konnte.
        RTS

_WINDOWNAME    DC.B   'CON:10/10/600/100/Mein Fenster',0
_WINDOWHANDLE  DC.L   0
```

Die Routine zum Schließen des Fensters:

```
WINDOW_CLOSE
        MOVE.L    _WINDOWHANDLE(PC),D1
        CALLDOS   Close
        RTS
```

Beachten Sie, daß das geöffnete Fenster auf dem Workbench-Screen hinter dem ASM-ONE-Screen erscheint, den Sie mit der Maus herunterziehen müssen.

Mit »CON:«-Fenstern können Sie schon einiges machen. Die folgende Routine gibt beispielsweise Text in das Fenster aus.

```
TEXT_WRITE
        MOVE.L   #TEXT,D2   ; Pointer auf Text
        MOVE.L   #TEXT_END-TEXT,D3   ; Textlänge
        MOVE.L   _WINDOWHANDLE,D1    ; Window-
                                     ; Handle
        CALLDOS  Write      ; Text schreiben
        RTS

TEXT    DC.B     'Hallo, dies ist ein Text'
TEXT_END
```

Programme, die alle diese Funktionen nutzen, finden Sie in den Beispielen »WindowExample.S« oder »Directory.S«.

11.3 Hardware-Programmierung

Wie bereits im ersten Kapitel erwähnt, ist einer der Gründe, Maschinensprache einzusetzen, der sehr einfache und direkte Zugriff auf die Hardware. Dieses Kapitel über die Hardware-Programmierung wird sich auf eine Einführung beschränken, für weitere Informationen sollten Sie auf das »Amiga Hardware Reference Manual« von Commodore zurückgreifen.

Sie werden das Betriebssystem in den meisten Fällen abschalten müssen, wenn Sie die Hardware direkt programmieren wollen.

Damit verhindern Sie, daß merkwürdige Effekte auftreten, wenn das Betriebssystem auch auf die Hardware zugreifen möchte.

Natürlich können Sie auch mit Betriebssystemroutinen alles Machbare programmieren, doch manche Programmierer ziehen es vor, genau zu wissen, was die Maschine tut. Wenn Sie die Chips direkt ansprechen, können Sie Ihre eigenen, hochoptimierten Routinen schreiben, die eventuell deutlich schneller als die des Betriebssystems sind. Allerdings ist dann nicht sicher, daß Ihr Programm auch noch auf zukünftigen Versionen der Hardware läuft.

Andererseits sind einige Erfahrungen in der Programmierung der Hardware aber auch dann nützlich, wenn Sie mit dem Betriebssystem programmieren wollen, denn sie sind eine große Hilfe beim Verständnis der exakten Funktion der Betriebssystemroutinen.

Auf der ASM-ONE-Diskette befindet sich ein Programmtext namens »NonSystemStartup.S«, der Routinen enthält, die das Betriebssystem komplett abschalten. Sie können programmieren, ohne sich über irgendwelche Nebeneffekte Gedanken machen zu müssen.

Nach einem Druck auf die linke Maustaste versetzt »NonSystemStartup.S« das System wieder in seinen vorherigen Status.

Die nun folgende Liste enthält alle Hardware-Register in Form ihrer Offsets zur Basisadresse »$DFF000«. Wenn Sie also beispielsweise »BplCon0« ansprechen wollen, müssen Sie dazu die Adresse »$DFF100« verwenden.

NAME	ADD	R/W	Function
BLTDDAT	$000	ER	Blitter destination early read
DMACONR	$002	R	DMA control read
VPOSR	$004	R	Read Vert Most sig. bit
VHPOSR	$006	R	Read vert and horiz pos of beam
DSKDATR	$008	ER	Disk data early read
JOY0DAT	$00A	R	Joy-Mouse 0 data
JOY1DAT	$00C	R	Joy-Mouse 1 data
CLXDAT	$00E	R	Collision data reg (read & clear)
ADKCONR	$010	R	Audio, disk control read
POT0DAT	$012	R	Pot counter pair 0 data
POT1DAT	$014	R	Pot coubter pair 1 data
POTINP	$016	R	Pot pin data read
SERDATR	$018	R	Serial port data and status read
DSKBYTR	$01A	R	Disk data byte and status read
INTENAR	$01C	R	Interrupt enable bits read
INTREQR	$01E	R	Interrupt request bits read
DSKPT	$020	W	Disk pointer (longword)
DSKLEN	$024	W	Disk length
DSKDAT	$026	W	Disk DMA data write
REFPTR	$028	W	Refresh pointer
VPOSW	$02A	W	Write vert most sig bit
VHPOSW	$02C	W	Write vert and horiz position
COPCON	$02E	W	Coprocessor control register
SERDAT	$030	W	Serial Port data and stop bits write
SERPER	$032	W	Serial port period and control
POTGO	$034	W	Pot count start
JOYTEST	$036	W	Write to all 4 mouse counters at once
STREQU	$038	S	Strobe for horiz sync (VB & equ)
STRVBL	$03A	S	Strobe for horiz sync (VB)
STRHOR	$03C	S	Strobe for horiz sync
STRLONG	$03E	S	Strobe for long horiz. line
BLTCON0	$040	W	Blitter control register 0
BLTCON1	$042	W	Blitter control register 1
BLTAFWM	$044	W	Blitter first word mask for source A
BLTALWM	$046	W	Blitter last word mask for source A
BLTCPT	$048	W	Blitter source C pointer (longword)

NAME	ADD	R/W	Function
BLTBPT	$04C	W	Blitter source B pointer (longword)
BLTAPT	$050	W	Blitter source A pointer (longword)
BLTDPT	$054	W	Blitter dest. D pointer (longword)
BLTSIZE	$058	W	Blitter start and size (width,higth)
BLTCMOD	$060	W	Blitter source C modulo
BLTBMOD	$062	W	Blitter source B modulo
BLTAMOD	$064	W	Blitter source A modulo
BLTDMOD	$066	W	Blitter dest. D modulo
BLTCDAT	$070	W	Blitter source C data
BLTBDAT	$072	W	Blitter source B data
BLTADAT	$074	W	Blitter source A data
DSKSYNC	$07E	W	Disk sync pattern register
COP1LC	$080	W	Copper pointer 1 (longword)
COP2LC	$084	W	Copper pointer 2 (longword)
COPJMP1	$088	W	Restart at pointer 1
COPJMP2	$08A	W	Restart at pointer 2
COPINS	$08C	W	Copper inst. fetch identify
DIWSTRT	$08E	W	Display window start (vert-hor)
DIWSTOP	$090	W	Display window start stop (vert-hor)
DDFSTRT	$092	W	Bit plane data fetch start
DDFSTOP	$094	W	Bit plane data fetch stop
DMACON	$096	W	DMA control write
CLXCON	$098	W	Collision control
INTENA	$09A	W	Interrupt enable bits
INTREQ	$09C	W	Interrupt request bits
ADKCON	$09E	W	Audio, disk, UART control
AUD0	$0A0		Audio channel 0 start address
AUD1	$0B0		Audio channel 1 start address
AUD2	$0C0		Audio channel 2 start address
AUD3	$0D0		Audio channel 3 start address
AUDxPTR	+$00	W	Audio channel x pointer
AUDxLEN	+$04	W	Audio channel x length
AUDxPER	+$06	W	Audio channel x period
AUDxVOL	+$08	W	Audio channel x volume
AUDxDAT	+$0A	W	Audio channel x data

NAME	ADD	R/W	Function
BPL1PT	$0E0	W	Bitplane 1 pointer
BPL2PT	$0E4	W	Bitplane 2 pointer
BPL3PT	$0E8	W	Bitplane 3 pointer0
1BPL4PT	$0EC	W	Bitplane 4 pointer
BPL5PT	$0F0	W	Bitplane 5 pointer
BPL6PT	$0F4	W	Bitplane 6 pointer
BPLCON0	$100	W	Bit plane ctrl reg (misc ctrl bits)
BPLCON1	$102	W	Bit plane ctrl reg (scroll value)
BPLCON2	$104	W	Bit plane ctrl reg (priority ctrl)
BPL1MOD	$108	W	Bit plane modulo (odd planes)
BPL2MOD	$10A	W	Bit plane modulo (even planes)
BPL1DAT	$110	W	Bit plane 1 data (parallel to serial)
BPL2DAT	$112	W	Bit plane 2 data (parallel to serial)
BPL3DAT	$114	W	Bit plane 3 data (parallel to serial)
BPL4DAT	$116	W	Bit plane 4 data (parallel to serial)
BPL5DAT	$118	W	Bit plane 5 data (parallel to serial)
BPL6DAT	$11A	W	Bit plane 6 data (parallel to serial)
SPR0PT	$120	W	Sprite 0 pointer
SPR1PT	$124	W	Sprite 1 pointer
SPR2PT	$128	W	Sprite 2 pointer
SPR3PT	$12C	W	Sprite 3 pointer
SPR4PT	$130	W	Sprite 4 pointer
SPR5PT	$134	W	Sprite 5 pointer
SPR6PT	$138	W	Sprite 6 pointer
SPR7PT	$13C	W	Sprite 7 pointer
SPR0	$140		Sprite 0 area
SPR1	$148		Sprite 1 area
SPR2	$150		Sprite 2 area
SPR3	$158		Sprite 3 area
SPR4	$160		Sprite 4 area
SPR5	$168		Sprite 5 area
SPR6	$170		Sprite 6 area
SPR7	$178		Sprite 7 area

NAME	ADD	R/W	Function
SPRxPOS	+$00	W	Sprite x Vert-Horiz position
SPRxCTL	+$02	W	Sprite x stop pos & control
SPRxDATAA	+$04	W	Sprite x image data a
SPRxDATAB	+$08	W	Sprite x image data b
COLOR0	$180	W	Color 0
COLOR1	$182	W	Color 1
bis			
COLOR31	$1BE	W	Color 31
SPECIAL	$1DC	W	NTSC control on Fatter Agnus

Zum Verständnis des folgenden Programmierbeispiels sollten Sie parallel ein Hardware-Referenzhandbuch zur Verfügung haben (im Anhang befindet sich ein Literaturverzeichnis).

Bevor wir mit Non-System-Programmen beginnen, möchte ich Ihnen anhand einiger Beispiele die direkte Abfrage der Hardware erläutern.

Eine der ersten und einfachsten Routinen ist hier die Routine zum Abfragen der Maustasten und des Joystick-Buttons.

Dazu müssen wir noch ein weiteres Register betrachten. Es befindet sich an Adresse »$BFE001« und weist folgende Belegung auf:

Bit	Function
7	Game port 1 (fire button)
6	Game port 0 (fire button)
5	Disk ready
4	Disk track 00
3	Write protect
2	Disk change
1	Led light (0 = filter on)
0	Memory overlay

Bit 6 ist der linke Mausknopf und Bit 7 der Joystick-Button. Wenn das jeweilige Bit 0 ist, bedeutet das, daß der entsprechende Knopf gedrückt ist.

```
LeftMouseButton:
        BTST    #6,$BFE001
        BEQ     LeftMousePressed
JoyButton:
        BTST    #7,$BFE001
        BEQ     JoyPressed
```

Um die rechte Maustaste abzufragen, müssen wir Bit 10 in »POTINP« an Adresse »$DFF016« testen:

```
RightButtonTest:
        BTST    #10,$DFF016
        BEQ     RightMousePressed
```

Wenn Sie Bits in Registern setzen oder testen, sollten Sie daran denken, daß »BSET«, »BTST« usw. nur auf Byte-Basis arbeiten. »BTST #10,??« entspricht also in Wirklichkeit »BTST #10-8,??«.

Wenn Sie Non-System-Programme schreiben, also Programme, die keine Library- und Kickstart-Routinen nutzen, empfehle ich Ihnen, »NonSystemStartup.S« zu verwenden. Dieses Beispiel stellt Ihnen eine einfach zu verwendende Standard-Routine zur Verfügung. Ein Beispiel zur Verwendung dieses Startup-Moduls finden Sie in »ScrollExample.S«.

Die Startup-Routine basiert auf einem Interrupt, der, »Vertical Blanking Interrupt« (»VBI«) genannt, jedesmal aktiviert wird, wenn das Bild neu aufgebaut wird. Das geschieht fünfzigmal in der Sekunde (auf NTSC-Amigas sechzigmal in der Sekunde). Man sagt, daß der Computer 50 Bilder pro Sekunde anzeigt. (Lesen Sie hierzu auch im Hardware-Handbuch über die Playfield-Hardware nach.)

Mit diesem Interrupt können Sie eine oder mehrere Routinen starten. In dem bereits erwähnten Scroll-Beispiel wird der Text auf dem Bildschirm einmal pro Bild bewegt.

In einer Sekunde werden Sie den Scroll-Text also an 50 verschiedenen Positionen sehen, jedesmal ein wenig nach links verschoben. Da das Auge wechselnde Bilder nur bis zirka 25 Bilder pro Sekunde auseinanderhalten kann, erscheinen 50 Bilder pro Sekunde als fließende Animation.

Diese Technik kann auch mit anderen Routinen kombiniert werden. Sie könnten beispielsweise einen Ball zeichnen, ihn im nächsten Bild löschen und etwas versetzt neu zeichnen. Auf diese Weise ist es möglich, einen hüpfenden Ball darzustellen. Oder auch eine kleine Animation, wenn Sie den Ball selbst in jedem Bild etwas verändert zeichnen.

Damit diese Routinen einwandfrei arbeiten, müssen Sie sie selbstverständlich an der richtigen Stelle in das »NonSystemStartup«-Modul einfügen.

Suchen Sie zunächst diese Zeilen:

```
INTER:
      MOVEM.L   D0-D7/A0-A6,-(A7)     ; Put
                                  ; registers on stack
      LEA.L     $DFF000,A6
      MOVE.L    #SCREEN,$E0(A6)

;--- Place your interrupt routine here ---

      MOVE.W    #$4020,$9C(A6)        ; Clear
                                  ; interrupt request
      MOVEM.L   (A7)+,D0-D7/A0-A6     ; Get regis-
                                     ; ters from stack
      RTE
```

Fügen Sie die Routine nach dem Text »;--- Place ...« ein.

Objekte auf den Schirm zu zeichnen braucht Zeit. Da das Objekt fünfzigmal pro Sekunde neu gezeichnet werden muß, um einen weichen Ablauf zu gewährleisten, darf die Routine natürlich nicht mehr als 1/50 Sekunde Zeit in Anspruch nehmen. Wenn eine Routine mehr Zeit braucht, sollte sie nicht aus dem Interrupt heraus gestartet werden.

Nehmen wir an, daß Sie eine Routine erstellt haben, die ein Bild berechnet, dafür aber 30 Sekunden benötigt. Dann würden Sie diese zwischen den folgenden Stellen in den Programmtext einfügen:

```
;***** Your main routines *****
;***** Main loop Test mouse button *****
```

Sie sollten erst Programme, die direkt auf die Custom-Chips zugreifen, schreiben, wenn Sie einerseits die Maschinensprache beherrschen und zum anderen die Funktionsweise der Custom-Chips verstehen.

Eine genaue Einführung in dieses weite Feld kann hier nicht gegeben werden, da die Amiga-Hardware sehr weit entwickelt ist. Ich empfehle Ihnen, sich durch Literatur über diese Thematik näher zu befassen. Es folgt noch ein kleines Beispiel, das ein Objekt auf den Bildschirm zeichnet.

11.3.1 Erzeugung eines Objekts

Um Animationen auf dem Amiga zu programmieren, wird Ihnen der Blitter eine nützliche Hilfe sein. Er ist schnell und nicht sehr schwer zu programmieren. Bevor wir zu unserem Programmierbeispiel kommen, benötigen wir einen Überblick über die Register des Blitters. (Auch hier muß allen Offsets noch die Basisadresse »$DFF000« hinzuaddiert werden.)

NAME	ADD	R/W	Function
BLTCON0	$040	W	Blitter control register 0
BLTCON1	$042	W	Blitter control register 1
BLTAFWM	$044	W	Blitter first word mask for source A
BLTALWM	$046	W	Blitter last word mask for source A
BLTCPT	$048	W	Blitter source C pointer (longword)
BLTBPT	$04C	W	Blitter source B pointer (longword)
BLTAPT	$050	W	Blitter source A pointer (longword)
BLTDPT	$054	W	Blitter dest. D pointer (longword)
BLTSIZE	$058	W	Blitter start and size (width,higth)
BLTCMOD	$060	W	Blitter source C modulo
BLTBMOD	$062	W	Blitter source B modulo
BLTAMOD	$064	W	Blitter source A modulo
BLTDMOD	$066	W	Blitter dest. D modulo
BLTCDAT	$070	W	Blitter source C data
BLTBDAT	$072	W	Blitter source B data
BLTADAT	$074	W	Blitter source A data

Die Bits der beiden Kontrollregister sind wie folgt belegt:

Bit	BLTCON0	BLTCON1	Bit	BLTCON0	BLTCON1
15	ASH3	BSH3	07	LF7	x
14	ASH2	BSH2	06	LF6	x
13	ASH1	BSH1	05	LF5	x
12	ASH0	BSH0	04	LF4	EFE
11	USEA	x	03	LF3	IFE
10	USEB	x	02	LF2	FCI
09	USEC	x	01	LF1	DESC
08	USED	x	00	LF0	LINE

ASH0-3	- Source A shift value
BSH0-3	- Source B shift value
USEA-D	- Bit pattern to select source A - D.
LF0-7	- Logic function minterm select
EFE	- Exclusive fill enable
IFE	- Inclusive fill enable
FCI	- Fill carry input
DESC	- Descending bit
LINE	- Line mode (1=On)

Der Blitter verarbeitet drei Quellen (A, B, C) und ein Ziel (D). Im Prinzip nimmt der Blitter Datenworte aus den drei Quelladressen und errechnet aus diesen nach einem wählbaren Algorithmus das Zielwort. Dieser Algorithmus wird durch das LF-Byte (Bit 0 bis 7 in »BLTCON0«) festgelegt.

Angenommen, wir haben ein Objekt (Pattern) und einen Screen mit der Breite von 40 Byte. Wir können Quelle A als Zeiger auf die Objektdaten verwenden und das Ziel D als Zeiger auf die Bildschirmposition, an der das Objekt erscheinen soll. Wenn wir jedoch nur diese beiden hätten, würden wir einfach den Block kopieren, doch wir wollen den vorigen Inhalt des Bildschirms ja nicht löschen, sondern nur das Objekt einfügen. Also laden wir die Quelle C mit derselben Adresse wie in Ziel D, damit an Stellen (Pixeln), die im neuen Objekt nicht verwendet werden, wieder der alte Hintergrund erscheint. Die folgenden drei Zeilen erledigen das:

```
MOVE.L   #OBJECT,$DFF050     ; Quelle A
MOVE.L   #SCREEN,$DFF048     ; Quelle C
MOVE.L   #SCREEN,$DFF054     ; Ziel D
```

Wenn das Objekt ein Wort breit ist (2 Bytes), müssen wir den Modulo auf der Quelle C und dem Ziel D auf 38 (40 minus 2) setzen. Der Modulo-Wert von Quelle A muß dann 0 sein.

```
MOVE.W   #0,$DFF064    ; Modulo Quelle A
MOVE.W   #38,$DFF060   ; Modulo Quelle C
MOVE.W   #38,$DFF066   ; Modulo Ziel D
```

Nun setzen wir die Kontrollregister, beginnend mit »BLTCON0«. Der Shiftwert wird auf 0 (»%0000«) gesetzt, wir verwenden A, C und D, also müssen die nächsten vier Bits auf »%1011« gesetzt werden. Das Byte für die logische Funktion wird auf »%11111010« gesetzt. Wenn der Blitter die drei Quellen verknüpft, betrachtet er immer ein Bit zur Zeit. Drei Bits geben maximal acht Kombinationen, daher existieren acht LF-Bits. Diese Bits werden gemäß der folgenden Liste gesetzt.

Bit	A	B	C	Erwünschtes Ergebnis in D
7	1	1	1	1
6	1	1	0	1
5	1	0	1	1
4	1	0	0	1
3	0	1	1	1
2	0	1	0	0
1	0	0	1	1
0	0	0	0	0

Wir verwenden die Quelle B hier nicht, müssen also nur über die vier verschiedenen Kombinationen aus A und C nachdenken (11 = 1, 10 = 1, 01 = 1, 00 = 0). Für jedes Bit gilt: Wenn ein Bit im Objektwort und im Hintergrundwort gesetzt ist, soll das entsprechende Bit im Zielwort auch gesetzt werden ... etc.

Das Ergebnis der obigen Überlegungen ist, daß »BLTCON0« wie folgt gesetzt werden muß:

```
MOVE.W   #%0000101111111010,$DFF040
```

Kontrollregister 1 muß auf 0 gesetzt werden

```
MOVE.W   #%0000000000000000,$DFF042
```

Um den Blitter zu starten, müssen Sie die Größe des Objektes im Register »BLTSIZE« eintragen. Die Größe berechnet sich als 64 mal Höhe plus Breite geteilt durch 2.

```
MOVE.W   #64*8+2/2,$DFF058
```

Diese Zeilen können wir nun in einem kleinen Programm zusammenfügen:

```
MAKE_OBJECT:
        LEA.L    $DFF000,A6       ; Basisadresse
                                ; Customchips nach A6

WAIT:   BTST     #14-8,$2(A6)           ; Ist der
                                  ; Blitter bereit?
        BNE.B    WAIT             ; sonst auf
                                  ; Blitter warten

        MOVE.W   #%0000101111111010,$40(a6)
        MOVE.W   #%0000000000000000,$42(a6)

        MOVE.L   #OBJECT,$50(a6)        ; Quelle A
        MOVE.L   #SCREEN,$48(a6)        ; Quelle C
        MOVE.L   #SCREEN,$54(a6)        ; Ziel D
```

```
         MOVE.W   #0,$64(a6)      ; Modulo Quelle A
         MOVE.W   #38,$60(a6)     ; Modulo Quelle C
         MOVE.W   #38,$66(a6)     ; Modulo Ziel D

         MOVE.L   #$FFFFFFFF,$44(a6)  ; Keine
                                  ; Maskierung
         MOVE.W   #64*8+2/2,$58(a6)   ; Blitter
                                      ; starten
         RTS

OBJECT:  DC.W     %0000000110000000   ;Objektdaten
         DC.W     %0000001111000000
         DC.W     %0100011001100010
         DC.W     %0100111001110010
         DC.W     %0111111111111110
         DC.W     %0100011111100010
         DC.W     %0000001111000000
         DC.W     %0000011111100000
```

Weitere Beispiele zur Nutzung des Blitters finden Sie in den Programmen »ScrollExample.S« und »LineExample.S«.

Anhänge

A. Voreinstellungen

Um sicherzustellen, daß Sie ASM-ONE so konfigurieren können, wie Sie möchten, wurde eine Vielzahl von Optionen implementiert. Sie finden folgende Optionen in den Menüs »Project« und »Assembler«.

Option	Voreinstellung	Abkürzung
Rescue	Aus	RS
Level 7	Aus	L7
NumLock	Ein	NL
AutoAlloc	Ein	AA
ReqLibrary	Aus	RL
PrinterDump	Aus	PD
Interlace	Aus	IL
1 Bitplane	Aus	1B
Source .S	Ein	.S
LineNumbers	Aus	LN
AutoIndent	Ein	AI
ShowSource	Ein	SS
ListFile	Aus	LF
Pageing	Ein	PG
Halt Page	Ein	HP
All Errors	Aus	AE
Debug	Aus	DB
Label:	Aus	L:
UCase=LCase	Ein	UL
Offset(A4)	Aus	A4
DisAssemble	Ein	DA
OnlyAscii	Aus	OA
Comment	Aus	C
Close WB	Aus	CW

Sie haben die Möglichkeit, die **Voreinstellungen** in einer Datei namens »ASM-One.Pref« auf »S:« zu **speichern**. Diese Datei besteht aus Plus- und Minuszeichen, jeweils gefolgt von der Abkürzung der jeweiligen Option.

Beispielsweise:

```
-RS-L7+NL+AA+RL etc.
```

Neben diesen Voreinstellungen können Sie in der Datei noch weitere Informationen unterbringen, zum Beispiel:

```
-RS-L7+NL+AA+RL\c\200\
```

Dies würde beim Start von ASM-ONE automatisch 200 KByte Arbeitsspeicher im Chip-Memory reservieren. Natürlich kann auch jeder andere Befehl nachfolgen. Das Zeichen »\« gilt als »Return«. Zur Festlegung einer Dateiextension beim Abspeichern ist das Zeichen »!« in der Voreinstellungsdatei erlaubt (siehe Option »Source.S«).

Beschreibung der Optionen:

Rescue (Retten)

Die Funktion »Rescue« **restauriert** Ihren **Programmtext**, wenn Ihr Programm abgestürzt ist, auch wenn das System abgeschaltet war. Ihr Programm übersteht so auch zum Beispiel eine Division durch null während einer Level-3-Interrupt-Routine.

WARNUNG: Diese Option kann das Multitasking-System Ihres Amiga beeinträchtigen. Aber wenn Sie Non-System-Programme schreiben und ein Absturz auftritt, müßten Sie andernfalls den Amiga neu starten und haben mit dieser Option wenigstens eine Chance, den Programmtext noch zu speichern.

Level 7

Die Option »Level 7« erlaubt Ihnen, Ihre **Programme** zu einem beliebigen Zeitpunkt **abzubrechen**. Nehmen wir zum Beispiel eine Endlosschleife. Derartige Programmierfehler können nur so umgangen und das Programm noch gespeichert werden. Um die Level-7-Option nutzen zu können, müssen Sie ein wenig Bastelarbeit leisten. Wenn Sie keine Kenntnisse in bezug auf Hardware-Basteleien besitzen, sollten Sie jemanden suchen, der sich in diesem Bereich auskennt, denn Sie können hier durch einen Fehler Ihren Computer irreparabel beschädigen!

Wenn Sie einen **Amiga 500** haben, gehen Sie wie folgt vor: An der linken Seite Ihres Computers befindet sich ein 86poliger Verbinder hinter einer Abdeckklappe. Nachdem Sie diese entfernt haben, sehen Sie eine Steckleiste die nach folgendem Schema belegt ist:

	GND	1		2		
		3		4		
		5		6		U
O		7		8		N
B		9		10		T
E		11		12		E
R		13		14		R
S		15		16		S
E						E
I		39		40	_IPL0	I
T		41		42	_IPL1	T
E		43		44	_IPL2	E
		83		84		
		85		86		

Nun benötigen Sie noch drei Dioden und einen Taster. Verbinden Sie je eine Diode mit der positiven Seite mit den drei IPL-Kontakten. Die negativen Enden der Dioden verbinden Sie miteinander und diese mit einem der Tasterkontakte. Der andere Tasterkontakt wird auf Masse (GND) gelegt.

40	_IPL0	**- Diode +**		
42	_IPL1	**- Diode +**	**-- / - Schalter**	**(1) GND**
44	_IPL2	**- Diode +**		

Wenn Sie einen **Amiga 2000** besitzen, gehen Sie folgendermaßen vor:

Öffnen Sie Ihren Rechner, und lokalisieren Sie die fünf 100poligen Expansion-Slots. Auf der Leiterplatte ist Pin 1 bezeichnet. Der Verbinder ist wie folgt belegt:

	GND	1		2		
		3		4		
		5		6		**U**
O		7		8		**N**
B		9		10		**T**
E		11		12		**E**
R		13		14		**R**
S		15		16		**S**
E						**E**
I		39		40	E-Int 7	**I**
T						**T**
E		97		98		**E**
		99		100		

Wenn Sie Pin 40 über einen Taster mit Pin 0 verbinden, lösen Sie einen Level-7-Interrupt aus.

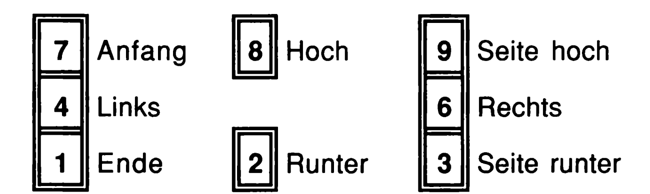
!!! Wichtig !!!

Wenn Sie sich nicht vollständig sicher sind, daß Sie diese Bastelanleitung nachvollziehen können, dann bauen Sie sie bitte **nicht** selbst nach. Falsche Verbindungen können schwerwiegende Schäden zur Folge haben.

Der DMV - Daten- und Medienverlag übernimmt keinerlei Garantien für die Richtigkeit der obigen Schaltung. Wenn Sie diese Schaltung an Ihren Amiga anschließen, handeln Sie auf eigenes Risiko.

NumLock

Mit dieser Option wird das **numerische Tastenfeld** wie folgt umdefiniert:



AutoAlloc

Wenn Sie zum Beispiel eine »**DATA_C**«**-Sektion erzeugen**, wird diese mit »AutoAlloc« im Chip-Memory erzeugt. Ohne »AutoAlloc« wird Ihr normaler Arbeitsbereich verwendet, ohne Rücksicht darauf zu nehmen, ob es sich bei diesem um Chip- oder Fast-Memory handelt.

ReqLibrary

Die »req.library« ist eine Library mit Funktionen, die den **File-Requester** und die restlichen Requester **erzeugt**. Diese Requester sind einfach anzuwenden, doch die Library braucht ungefähr 14 KByte Speicherplatz. Wenn Sie auf diese 14 KByte nicht verzichten können, können Sie diese Option desaktivieren. Wenn der Assembler aber bereits einmal **mit** der »req.library« gestartet wurde, wird der von ihr belegte Speicher erst beim nächsten Reset wieder freigegeben.

PrinterDump

Diese Option aktiviert die **Ausgabe** auf den **Drucker**. Wenn sie gesetzt ist, wird sämtlicher Text auch auf dem Drucker ausgegeben. Sie können diese Option auch mit »Amiga-P« umschalten. Wenn Sie den Drucker zum erstenmal ansprechen, wird zunächst noch das »printer.device« nachgeladen.

Interlace

Schaltet die **Anzeige** auf den **Interlace-Modus** um. Wenn für diese Aktion nicht ausreichend Speicher zur Verfügung steht, wird statt dessen ein Fenster mit einer Bitplane in mittlerer Auflösung geöffnet.

1 Bitplane

Schaltet die **Anzeige** auf nur eine **Bitplane** um. Dies spart ein wenig Speicher, aber der Cursor nimmt dieselbe Farbe wie der Text an und ist somit unter Umständen schwer zu finden.

Source .S

Fügt, wenn gesetzt, beim **Speichern** automatisch ein ».S« an den Dateinamen an. Sie können das ».S« im Einzelfall manuell löschen oder generell mit dieser Option ausschalten.

Wenn die Option »Source .S« eingeschaltet ist, steht Ihnen in der Preference-Datei die Option »!« zur Verfügung, mit der Sie beliebige Extensionen (Kennungen) beim Abspeichern an den Dateinamen anhängen können.

Syntax: **!.**<Extension (maximal drei Zeichen)>

Beispiel: `!.ASM`

Ist die Option nicht aktiviert, wird an die »SHOW«- beziehungsweise »HIDE«-Gadgets in der Requester-Library keine Erweiterung angehängt.

LineNumbers

Mit dieser Option wird **jede Zeile** im Editor mit einer **Zeilennummer** versehen. Ansonsten ändert sich an der Arbeitsweise nichts.

AutoIndent

Rückt jede neue **Zeile** automatisch ebenso weit **ein** wie die jeweils vorige. Dies ist nützlich, wenn Sie Ihren Programmtext stark mit Tabulatoren und ähnlichem strukturiert haben.

ShowSource

Muß gelöscht werden, wenn Sie **Programme**, deren Programmtext Sie nicht besitzen, mit der »Trace«-Funktion **analysieren** wollen. Die Option ist auch sehr nützlich, wenn Sie viele Makros verwenden und die bereits übersetzten Makros betrachten wollen.

ListFile

Wenn diese Option gesetzt ist, wird bei jedem Assembler-Lauf ein **Listfile erzeugt**. Dieses Listfile enthält Zeilennummern, Speicheradressen, Speicherinhalte und die jeweiligen Assembler-Befehle und am Ende die Symboltabelle.

Paging

Wenn Sie das **Listfile seitenweise** aufgeteilt haben möchten, können Sie das mit dieser Funktion erreichen.

Halt Page

Wenn das **Paging** aktiviert ist, können Sie mit dieser Option erreichen, daß der Rechner nach jeder Seite auf einen Tastendruck wartet.

All Errors

Normalerweise bricht ASM-ONE nach dem ersten Fehler den Assembliervorgang ab. Mit dieser Option durchläuft der Assembler in jedem Falle den gesamten Programmtext. Sie können mit dieser Option zum Beispiel eine **komplette Fehlerliste** zur Druckausgabe erzeugen, um die Fehler dann gesammelt zu korrigieren.

Debug

Wenn Sie Ihren Programmtext häufig debuggen, können Sie diese Option setzen, um Zeit zu sparen. Zwar brauchen Sie dadurch etwas mehr Speicher, doch muß der Debugger dann den **Programmtext nicht mehr neu assemblieren**.

Label:

Der ASM-ONE-Assembler ist kompatibel zum Makro-Assembler. Das bedeutet, daß Label nicht durch Doppelpunkte abgeschlossen werden müssen. Ein Befehl wird von einem Label allein dadurch unterschieden, daß ein Befehl nicht in der ersten Spalte beginnt.

Da einige Assembler für **Label** einen abschließenden **Doppelpunkt vorschreiben** und so ermöglichen, daß auch Befehle in der ersten Spalte beginnen können, wurde die Option »Label:« vorgesehen. Wenn Sie gesetzt ist, können Sie auch Programmtexte dieser (zumeist älteren) Assembler-Versionen assemblieren.

UCase=LCase

Manchmal möchten Sie dasselbe **Symbol zweimal verwenden**. Wenn Sie diese Option desaktivieren, ist es möglich, zwei gleiche Namen einfach durch unterschiedliche Groß- und Kleinschreibung zu trennen, zum Beispiel:

»DOSBase« und »DOSBASE«

Diese beiden Symbole werden normalerweise als ein und dasselbe Symbol betrachtet, doch bei desaktivierter Option sind die beiden Symbole für den Assembler unterschiedlich.

DisAssemble

Im Debugger ist es manchmal vorteilhaft, beim Ansehen des Programmtextes die **nächste Zeile disassembliert** zu sehen. Dadurch kann man einfacher feststellen, ob die derzeitige Zeile korrekt disassembliert wurde.

OnlyAscii

Wenn Sie einen ASCII-Dump betrachten und dabei bestimmte Texte auffinden möchten, stören fremde Sonderzeichen nur. Mit »OnlyAscii« werden nur normale Buchstaben, Ziffern und Zeichen angezeigt und **Sonderzeichen durch** einen **Punkt ersetzt**.

;Comment

Normalerweise ist ein **Kommentar** alles das, was nach einer korrekt beendeten Anweisung folgt. Eine Anweisung wird mit einem oder mehreren Leerzeichen beendet. Wenn aber durch Zufall ein Leerzeichen im Operandenfeld eingefügt wurde, so behandelte der Assembler bisher den rechts nach dem Leerzeichen folgenden Teil wie einen Kommentar.

Beispiel:

```
MOVE.W   d0,DataPtr   +2
```

Hier wird »+2« wie ein Kommentar behandelt. Setzen Sie das Flag »;Comment«, um diese gemeinen Fehler zu vermeiden (im Preference-File »+;C«)

Close WB

Wenn Sie dies setzen, wird der **Workbench-Screen geschlossen**. Es ist jedoch nur möglich, die Workbench zu schließen, wenn kein Task-Window, beispielsweise das CLI-Fenster, geöffnet ist. Im Preference-File heißt diese Option »CW«.

B. Literaturempfehlungen

Dieses Handbuch wird nicht ausreichen, wenn Sie den Amiga und ASM-ONE wirklich ausreizen wollen. Die Bücher, die ich Ihnen empfehle, lassen sich in zwei Arten teilen: Amiga-spezifische Bücher und Bücher zur MC68000-Programmierung.

Hardware Reference Manual (Commodore)

Wenn Sie eine genaue Beschreibung der Amiga-Hardware suchen, dann ist dies genau das richtige Buch. Es enthält auch einige Programmierbeispiele. Für einen fortgeschrittenen Programmierer sollte es nicht allzu schwer sein, die Fakten in eigenen Programmen anzuwenden.

Libraries and Devices (Commodore)

Wenn Sie das Betriebssystem des Amiga ausnutzen möchten, sagt Ihnen dieses Buch nahezu alles über die verschiedenen Library-Funktionen. Unabhängig von Ihren Programmierabsichten, ist ein grundlegendes Verständnis des gesamten Amiga-Systems bei der Betriebssystemnutzung Voraussetzung. Assembler-Beispiele sind zwar rar, denn vorwiegend sind die Beispiele in C geschrieben, so daß Sie häufig mit den Assembler-Includes vergleichen müssen.

Includes and Autodocs (Commodore)

Wenn Sie »Libraries und Devices« bereits kennen, ist dieses Buch der nächste Schritt. Es ist detaillierter, schematischer und erklärt genauer, was bei den verschiedenen Funktionsaufrufen vor sich geht; zum Beispiel auch, in welchen Registern die Argumente übergeben werden müssen.

Amiga System Programmers Guide (Abacus/Data Becker)

Dieses Buch ist eine Mischung aus dem »Hardware Reference Manual« und »Libraries and Devices«, wobei der thematische Schwerpunkt auf »Hardware« liegt. Das Buch erläutert alles anhand von Beispielen, wobei die Fülle der Beispiele die Vielfalt der Themen begrenzt. Dieses Buch ist ein guter Ersatz für das »Hardware Manual«, doch wenn Sie ein ernsthafter Systemprogrammierer sein wollen, dann können Sie auf »Libraries and Devices« und »Includes and Autodocs« nicht verzichten. Wenn Sie nur **ein** umfangreiches Buch zum Thema Amiga-Programmierung durcharbeiten wollen, dann empfehle ich Ihnen dieses.

Amiga Machine Language (Abacus/Data Becker)

Dieses Buch beinhaltet eine Menge Themen über die Maschinensprache-Programmierung auf dem Amiga. Es basiert auf vielen Routinen zur Benutzung der Libraries. Fast alle Routinen können zusammen verwendet werden. Das Buch ist leicht verständlich geschrieben und preiswert. Wenn Sie an einer kleinen Einführung über die Erstellung von Windows, Menüs und Gadgets interessiert sind, dann kann ich Ihnen dieses Buch empfehlen. Es ist allerdings kein Ersatz für das »Hardware Manual«.

Programming the MC68000 (Sybex/Steve Williams)

Dies ist wirklich ein gutes Buch über die Programmierung des MC68000. Vor allem bietet es eine knappe Beschreibung aller Befehle und Adressierungsmodi. Zum Umfang gehören auch allgemeine Prozessor-Programmierbeispiele und Vorschläge zur Erstellung strukturierter Programme.

C. Fehlermeldungen

Bei der Assemblierung eines neu geschriebenen Programms werden Sie mit Sicherheit die eine oder andere Fehlermeldungen erhalten. Um einen Fehler zu korrigieren, ist die Fehlermeldung, die kurz erklärt, wo der Fehler liegt, von großem Nutzen. Hier folgt eine detailliertere Beschreibung der verschiedenen Fehlermeldungen von ASM-ONE.

1. Workspace Memory full

Der Assembler braucht für Labels, Reloc-Tabellen, Opcodes etc. zusätzlichen Speicher, den er sich vom Workspace abzweigt. Wenn nicht genug Speicherplatz reserviert wurde, erscheint diese Fehlermeldung.

2. Address Reg. Byte/Logic

Ein Adreßregister »(An)« kann nur beschränkt angesprochen werden. Sie können Adreßregister nicht als Bytes ansprechen, und Sie können sie auch nicht in logischen Operationen verwenden (wie »OR.W A1,D2«).

3. Address Reg. Expected

In einigen Adressierungsarten können Sie nur Adreßregister verwenden, zum Beispiel »(An)«, »(An)+«, »-(An)«, »nn(An,Rn)«. In manchen Befehlen, wie zum Beispiel »LEA <ea>,An«, können Sie auch nur ein Adreßregister als Ziel angeben.

4. Data Reg. expected

Entsprechend Fehlermeldung Nummer 3 bei Adreßregistern, gibt es auch verschiedene Befehle, die ein Datenregister als Operanden erwarten.

5. Comma expected

Wenn ein Befehl zwei oder mehr Operanden erwartet, müssen Sie die einzelnen Argumente durch Komma trennen. Manche Programmierer trennen Argumente gerne zusätzlich durch einen Tabulator, dieser muß dann aber immer hinter dem Komma stehen und nicht davor.

6. Double Symbol

Es ist nicht möglich, denselben Namen mehr als einmal zu verwenden. Die einzigen Symbole, die neu definiert werden können, sind die mit der Direktive »SET« definierten.

7. Unexpected end of file

Eine Makro-Definition, ein Wiederholungsblock oder eine If-Direktive müssen mit einem »ENDM«, »ENDR« beziehungsweise »ENDC« beendet werden. Wenn das Ende des Programms erreicht wird, bevor das entsprechende Ende der Konstruktion gefunden wurde, erscheint diese Fehlermeldung.

8. User made FAIL

ASM-ONE erkennt eine Direktive namens »FAIL« und gibt die Fehlermeldung Nummer 8 aus. Sie können dies verwenden, um zum Beispiel die Übergabeparameter für ein Makro zu testen und gegebenenfalls eine Fehlermeldung auszulösen.

9. Illegal Command

Ein unbekannter Befehl wurde eingegeben.

10. Illegal Address size

Sie können eine absolute Adresse explizit als Wort oder Langwort angeben, zum Beispiel »MOVE.L $4.W,A6«. Die beiden zulässigen Größen sind ».W« und ».L«, der Wertebereich ist einzuhalten. Wenn das nicht geschieht, erscheint diese Fehlermeldung.

11. Illegal Operand

Eine Zeile Programmtext läßt sich in verschiedene Felder aufspalten: Label, Befehl, Operanden, Kommentar. Bei fehlerhaften Operanden erscheint diese Fehlermeldung.

12. Illegal Operator

Weist auf einen fehlerhaften Befehl hin.

13. Illegal Section type

Es gibt drei Typen von Sektionen: »CODE«, »DATA« und »BSS«. Diese Typen können wie folgt erweitert werden:

CODE_P, CODE_C, CODE_F

DATA_P, DATA_C, DATA_F

BSS_P, BSS_C, BSS_F

Alle anderen Typen sind nicht zulässig und erzeugen diese Fehlermeldung.

14. Illegal Operator in BSS area

Es ist möglich, verschiedene Sektionen anzulegen (»CODE«, »DATA« und »BSS«, siehe oben). Eine »BSS«-Sektion wird nicht initialisiert, kann also keine definierten Werte aufnehmen. Daher sind alle codeproduzierenden Statements nicht zugelassen, auch nicht »DC« und »DCB«, und erzeugen diese Fehlermeldung. Erlaubt ist nur »DS«.

15. Illegal Order

Der Befehl »MOVEM« erwartet als Argument eine Registerliste, zum Beispiel »D0-D3/A4«. Die Register müssen in aufsteigender Reihenfolge mit den Datenregistern zu Beginn aufgezählt werden.

16. Illegal reg. size

Die Adressierungsart »nn(An,Rn)« erlaubt eine optionale Größenangabe auf dem Indexregister »Rn«, diese kann nur ».W« oder ».L« sein, zum Beispiel »10(A3,D0.W)«, andernfalls erfolgt diese Fehlermeldung.

17. Illegal Size

Einige Befehle arbeiten mit einer Größenangabe (».B«, ».W« oder ».L«). Aber nicht alle können alle drei Größen verarbeiten. Bei einer falschen Größenangabe tritt diese Fehlermeldung auf.

18. Illegal macro def.

Es ist nicht möglich, ein Makro innerhalb eines Makros zu definieren, auch wenn Makros innerhalb von Makros aufgerufen werden können.

19. Immediate operand ex.

Dieser Fehler tritt auf, wenn Sie zum Beispiel einem »ADDI«- oder »ADDQ«-Befehl keine Konstante, sondern ein Register als ersten Parameter übergeben.

20. Include Jam

Eine sehr seltene Fehlermeldung, die nur dann auftreten kann, wenn Sie eine Datei erst in »Pass2« einbinden.

21. Macro overflow

Sie können Makros bis zu einer Verschachtelungstiefe von 25 aus anderen Makros heraus aufrufen. Diese Restriktion ist in der limitieren Stapelkapazität begründet.

22. Conditional overflow

Auch konditionale Abfragen können nur maximal 25fach verschachtelt werden. Auch hier liegt der Grund in der limitierten Stapelkapazität.

23. Section overflow

Sie können nur maximal 255 Sektionen verwenden. Diese Begrenzung begründet sich aus der Speicherkapazität.

24. Include overflow

Sie können Include-Dateien nur maximal fünffach verschachteln. Die normalen Include-Dateien erreichen allerdings nur einen Include-Level von 3. Diese Restriktion hängt ebenfalls mit dem begrenzten Stapelspeicherplatz zusammen.

25. Repeat overflow

Sie können »REPT..ENDR«-Folgen nur maximal vierfach verschachteln. Grund hierfür ist ebenfalls die beschränkte Speicherkapazität des Stapels.

26. Double definition

Es ist nicht möglich, mit »BASEREG« dasselbe Register mehr als einmal als Basisregister zu definieren.

27. Invalid Addressing Mode

Dies ist der generelle Adressierungsfehler. Wenn Sie diese Fehlermeldung erhalten, ist entweder die Adressierungsart des ersten oder die des zweiten Operanden falsch oder nicht zulässig.

28. LOAD without ORG

Sie können Ihr Programm mit dem Statement »ORG« an eine absolute Adresse assemblieren lassen. Mit dem Statement »LOAD« modifizieren Sie nicht die Position, aber die

Startadresse, auf die sich alle absoluten Referenzen beziehen. Es macht keinen Sinn, ein »LOAD«-Statement zu verwenden, ohne mit »ORG« eine Zieladresse festgelegt zu haben.

29. Missing Quote

Wenn Sie Text-Strings verwenden, die mit einem der Anführungszeichen (' " `) beginnen, müssen Sie den Text mit demselben Zeichen beenden. Um das Zeichen im String selbst zu verwenden, müssen Sie es doppelt schreiben.

30. NO operand space allowed

Es ist nicht möglich, Tabulatoren beziehungsweise Leerstellen in einem Operanden zu verwenden.

31. NOT a constant/Label

Tritt auf, wenn in einem arithmetischen Ausdruck statt Labels und Konstanten versehentlich ein Makro-Name verwendet wurde.

32. Not in Repeat area

»ENDR« ist nur nach einleitendem »REPT« zulässig.

33. Not in macro

»ENDM«, »MEXIT« und »CMEXIT« sind nur in Makros zulässig.

34. Out of Range 0 bit

Diese Fehlermeldung wird nur vom Befehl zum Schieben von Speicherzellen ausgelöst. Speicherzellen können immer nur um ein Bit verschoben werden. Wenn Sie mehr schieben, erhalten Sie diese Fehlermeldung.

35. Out of Range 3 bit

Einige Befehle erlauben nur 3-Bit-Parameter (»ADDQ«, »SUBQ«), die Werte von 1 bis 8 annehmen können. Andere Werte sind nicht zulässig.

36. Out of Range 4 bit

Das Kommando »TRAP« enthält einen 4-Bit-Wert, welcher Trap auszulösen ist. Zulässig sind nur Werte von 0 bis 15.

37. Out of Range 8 bit

Befehle wie »MOVEQ« erlauben nur 8-Bit-Werte, die als »signed byte« betrachtet werden, also einen Wertebereich von -128 bis 127 haben. Andere 8-Bit-Werte dürfen Werte von -256 bis 255 annehmen.

38. Out of Range 16 bit

»Signed Words« haben einen Wertebereich von -32768 bis 32767. Normale Worte dürfen Werte von -65536 bis 65535 enthalten.

39. Relative Mode Error

Wenn Sie versuchen, einen relativen Wert als eine Konstante zu verwenden (außer in Adreßdistanzberechnungen), erhalten Sie diese Fehlermeldung.

40. Reserved Word

Sie können für Labels und Makros fast alle Namen verwenden. Ausgenommen sind lediglich die Namen der Register, um entscheiden zu können, ob nun das Register oder der Wert des Symbols gemeint ist, wenn der Registername in einem Befehl auftaucht.

41. Right parenthes Expected

Sie haben eine schließende Klammer vergessen.

42. String expected

Wenn eine Direktive einen String als Argument erwartet, muß auch ein solcher übergeben werden.

43. Undefined Symbol

Die Verwendung eines nicht definierten Symbols löst diese Fehlermeldung aus. Die einzigen immer definierten Symbole sind die Register und »NARG« (siehe bei der »MACRO«-Direktive).

44. Register Expected

Einige Befehle können nur mit Registern als Argumenten arbeiten. Wenn Sie etwas anderes angeben, erhalten Sie diese Fehlermeldung.

45. Word at Odd Address

Bestraft den Versuch, an einer ungeraden Adresse einen Befehl zu assemblieren, da der MC68000 keine Befehle an ungeraden Adressen ausführen kann.

46. Not local area

Lokale Labels können erst nach globalen Labels definiert werden. Dieser Fehler tritt auf, wenn Sie in Ihrem Programmtext als erstes ein lokales Label verwenden.

47. Code moved during pass 2

Dies ist ein kritischer Fehler, denn in »Pass1« werden alle Label-Werte ermittelt. Wenn sich zwischen »Pass1« und »Pass2« daran etwas ändert, ging irgend etwas schief. Dieser Fehler kann auftreten, wenn Sie konditionale Direktiven falsch verwenden.

48. Bcc.B out of range in Macro

Normalerweise werden »Bcc.B«-Befehle in ».W« umgewandelt, wenn der Branch den Offset-Bereich überschreitet. In Makros wird diese Umwandlung nicht vorgenommen und löst diese Fehlermeldung aus. Sie können so für bestimmte Branches sicherstellen, daß sie »byte-sized« sind, wenn Sie sie in ein Makro einbetten.

49. Out of range (20 to 100)

»PLEN« akzeptiert nur Werte zwischen 20 und 100.

50. Out of range (60 to 132)

»LLEN« akzeptiert nur Werte zwischen 60 und 132.

51. Linker limitation

Sie können keine Kalkulationen mit zwei extern referenzierten Symbolen durchführen.

Fehlermeldungen, die bei der Eingabe beziehungsweise Ausgabe auftreten können:

52. File Error

Tritt auf, wenn die Datei entweder nicht gefunden wurde oder ein Fehler in der Datei das korrekte Laden verhindert hat.

53. No Files

Tritt auf, wenn das Directory einer leeren Disk mit »V« (View) angezeigt werden soll.

54. No Object

Tritt auf, wenn Sie einen Programmtext mit »XREF« und »XDEF« als Objekt-File speichern wollen oder den Programmtext seit der letzten Assemblierung geändert haben.

55. No File Space

Die Diskette ist voll.

56. Printer Device Mission

Wenn Sie etwas ausdrucken wollen, muß der Amiga das »printer.device«, den Port-Handler und das »parallel.device« beziehungsweise das »serial.device« von Diskette nachladen. Wenn eine der Dateien fehlt, tritt diese Fehlermeldung auf.

57. Req.Library not found

ASM-ONE nutzt die »req.library«. Wenn diese Library nachgeladen werden soll, aber nicht vorhanden ist, tritt diese Fehlermeldung auf. Das kann passieren, wenn Sie ASM-ONE nicht gebootet haben und vergaßen, die »req.library« auf Ihre Boot-Diskette zu kopieren.

58. Illegal Path

Diese Fehlermeldung erhalten Sie, wenn Sie dem Befehl »V« (View) ein nicht vorhandenes Directory angeben.

59. Illegal Device

Sie können nur vier Diskettenlaufwerke ansprechen (0 bis 3). Alle anderen Laufwerksangaben sind nicht zulässig.

60. Write Protected

Es ist nicht möglich, auf eine schreibgeschützte Diskette zu schreiben.

61. No disk in drive

Keine Diskette im Laufwerk.

62. Not done

Tritt auf, wenn eine Operation durch »ESC« abgebrochen wurde.

D. Taktzyklentabellen

Wenn Sie zeitkritische Passagen eines Programms optimieren wollen, finden Sie in diesem Anhang alles Notwendige. Die angegebenen Zeiten sind in Taktzyklen angegeben, und Sie müssen die Werte durch 7,09 MHz (Europa) beziehungsweise 7,16 MHz (USA) teilen, um sie in Sekunden umzurechnen, :

```
ExeTime = Cycles/7090000
```

Wenn Ihre Routine also zum Beispiel 10000 Zyklen braucht, wird sie in 0,014 Sekunden ausgeführt.

Zur Verwendung der Tabellen:

Suchen Sie zunächst Ihre Instruktion heraus, und stellen Sie sicher, daß Sie die richtige Größe und die richtige Adressierungsart besitzen. Wenn nach der Zyklenangabe ein »+« vermerkt ist, müssen Sie die Zeit, die zur Berechnung der effektiven Adresse nötig ist, noch addieren. Einige berechnete Ausführungszeiten:

```
ADD.L    D0,10(A1,D1.W)    = 12 + 14 = 26 Zyklen
MOVE.W   (A0)+,100(A2)     = 18 Zyklen
ADD.L    D0,D0             = 8 Zyklen
```

Tabelle D.1.
Zeit zur Berechnung der effektiven Adresse

<ea>	Byte/Word	Long
Dn	0	0
An	0	0
(An)	4	8
(An)+	4	8
-(An)	6	10
d16(An)	8	12
d8(An,Rn)	10	14
xxxx.W	8	12
xxxx.L	12	16
d16(PC)	8	12
d8(PC,Rn)	10	14
#xxxx	4	8

Tabelle D.2.
MOVE - Bytes und Worte

Src\Dest	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xxx.W	xxx.L
Dn	4	4	8	8	8	12	14	12	16
An	4	4	8	8	8	12	14	12	16
(An)	8	8	12	12	12	16	18	16	20
(An)+	8	8	12	12	12	16	18	16	20
-(An)	10	10	14	14	14	18	20	18	22
d16(An)	12	12	16	16	16	20	22	20	24
d8(An,Rn)	14	14	18	18	18	22	24	22	26
xxxx.W	12	12	16	16	16	20	22	20	24
xxxx.L	16	16	20	20	20	24	26	24	28
d16(PC)	12	12	16	16	16	20	22	20	24
d8(PC,Rn)	14	14	18	18	18	22	24	22	26
#xxxx	8	8	12	12	12	16	18	16	20

Tabelle D.3.
MOVE - Langworte

Src\Dest	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xxx.W	xxx.L
Dn	4	4	12	12	14	16	18	16	20
An	4	4	12	12	14	16	18	16	20
(An)	12	12	20	20	20	24	26	24	28
(An)+	12	12	20	20	20	24	26	24	28
-(An)	14	14	22	22	22	26	28	26	30
d16(An)	16	16	24	24	24	28	30	28	32
d8(An,Rn)	18	18	26	26	26	30	32	30	34
xxxx.W	16	16	24	24	24	28	30	28	32
xxxx.L	20	20	28	28	28	32	34	32	36
d16(PC)	16	16	24	24	24	28	30	28	32
d8(PC,Rn)	18	18	26	26	26	30	32	30	34
#xxxx	12	12	20	20	20	24	26	24	28

Tabelle D.4. Arithmetische, logische und Vergleichs-operationen

Opcode	Size	<ea>,An	<ea>,Dn	Dn,<ea>
ADD	B W	8+	4+	8+
	L	6+ **	6+ **	12+
AND	B W	-	4+	8+
	L	-	6+ **	12+
CMP	B W	6+	4+	-
	L	6+	6+	-
DIVS	W	-	158+ *	-
DIVU	W	- 1	40+ *	-
EOR	B W	-	4+	8+
	L	-	8+	12+
MULS	W	-	70+ *	-
MULU	W	-	70+ *	-
OR	B W	-	4+	8+
	L	-	6+ **	12+
SUB	B W	8+	4+	8+
	L	6+ **	6+ **	12+

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.
* Maximalwert
** Insgesamt 8 Zyklen, wenn »<ea>« Datenregister direkt ist.

Tabelle D.5.
Immediate-Befehle

Opcode	Size	#,Dn	#,An	#,<ea>
ADDI	B W	8	-	12+
	L	16	-	20+
ADDQ	B W	4	8	8+
	L	8	8	12+
ANDI	B W	8	-	12+
	L	16	-	20+
CMPI	B W	8	8	8+
	L	14	14	12+
EORI	B W	8	-	12+
	L	16	-	20+
MOVEQ	L	4	-	-
ORI	B W	8	-	12+
	L	16	-	20+
SUBI	B W	8	-	12+
	L	16	-	20+
SUBQ	B W	4	8	8+
	L	8	8	12+

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.

Tabelle D.6.
Instruktionen mit nur einem Operanden

Opcode	Size	Register	Memory
CLR	B W	4	8+
	L	6	12+
NBCD	B	6	8+
NEG	B W	4	8+
	L	6	12+
NEGX	B W	4	8+
	L	6	12+
NOT	B W	4	8+
	L	6	12+
Scc	B false	4	8+
	B true	6	8+
TAS	B	4	10+
TST	B W 4 4+		
	L	4	4+

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.

Tabelle D.7.
Shift und Rotate

Opcode	Size	Register	Memory
ASR	B W	6+2n	8+
ASL	L	8+2n	-
LSR	B W	6+2n	8+
LSL	L	8+2n	-
ROR	B W	6+2n	8+
ROL	L	8+2n	-
ROXR	B W	6+2n	8+
ROXL	L	8+2n	-

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.

Tabelle D.8.
Bitmanipulations-Instruktionen

Opcode	Size	Dn,Dn	Dn,<ea>	#n,Dn	#n,<ea>
BCHG	B	-	8+	-	12+
	L	8*	-	12*	-
BCLR	B	-	8+	-	12+
	L	10*	-	14*	-
BSET	B	-	8+	-	12+
	L	8*	-	12*	-
BTST	B	-	4+	-	8+
	L	6*	-	10*	-

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.
* Maximalwert

Tabelle D.9.
Branch and Trap Instruction Clock Periods

Opcode	Disp	Taken	Not Taken
Bcc	B	10	8
	L	10	12
BRA	B	10	-
	W	10	-
BSR	B	18	-
	W	18	-
DBcc	cc True	-	12
	cc False	10	14
CHK	-	43+*	8+
TRAP	-	34	-
TRAPV	-	34	4

\+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.
* Maximalwert

Tabelle D.10.
JMP, JSR, LEA, PEA und MOVEM

Opcode	Size	(An)	(An)+	-(An)	d16(An)	d8(An,Rn)	xx.W	xx.L	d16(PC)	d8(PC,Rn)
JMP	-	8	-	-	10	14	10	12	10	14
JSR	-	16	-	-	18	22	18	20	18	24
LEA	-	4	-	-	8	12	8	12	8	12
PEA	-	12	-	-	16	20	16	20	16	20
MOVEM	W	12+4n	12+4n	-	16+4n	18+4n	16+4n	20+4n	16+4n	18+4n
M-R	L	12+8n	12+8n	-	16+8n	18+8n	16+8n	20+8n	16+8n	18+8n
MOVEM	W	8+4n	-	8+4n	12+4n	14+4n	12+4n	16+4n	-	-
R-M	L	8+8n	-	8+8n	12+8n	14+8n	12+8n	16+8n	-	-

n ist die Zahl der zu kopierenden Register

Tabelle D.11. Mehrfachgenaue Instruktionen

Opcode	Size	Dn,Dn	M,M
ADDX	B W	4	18
	L	8	30
CMPM	B W	-	12
	L	-	20
SUBX	B W	4	18
	L	8	30
ABCD	B	6	18
SBCD	B	6	18

Tabelle D.12. Diverses

Opcode	Size	Rn	<ea>	Rn,<ea>	<ea>,Rn
MOVE SR,?	-	6	8+	-	-
MOVE ?,CCR	-	12	12+	-	-
MOVE ?,SR	-	12	12+	-	-
MOVEP	W	-	-	16	16
	L	-	-	24	24
EXG	-	6	-	-	-
EXT	W	4	-	-	-
	L	4	-	-	-
LINK	-	16	-	-	-
MOVE USP,?	-	4	-	-	-
MOVE ?,USP	-	4	-	-	-
NOP	-	4	-	-	-
RESET	-	132	-	-	-
RTE	-	20	-	-	-
RTR	-	20	-	-	-
RTS	-	16	-	-	-
STOP	-	4	-	-	-
SWAP	-	4	-	-	-
UNLK	-	12	-	-	-

+ Addieren Sie die Zeit zur Berechnung der effektiven Adresse.

Tabelle D.13. Ausnahmebehandlungen

Exception	Periods
Address Error	50
Bus Error	50
Interrupt	44*
Illegal Instruction	34
Privileged Instruction	34
Trace	34

* Der Interrupt-Bestätigungszyklus wird als 4 Zyklen angenommen.

Indexverzeichnis

D

ASM-One

Professionelles Assembler-Entwicklungs-system für Commodore Amiga

»ASM-One«, das Entwicklungsspaket, das für den Commodore Amiga neue Maßstäbe setzt.

Der Editor/Assembler entspricht den Anforderungen, die an ein professionelles System gestellt werden: Superschnell, mit Block- und Makrofunktionen ausgestattet, erlaubt er eine übersichtliche Eingabe der Quelltexte. Schnelle Kopierroutinen erleichtern den Umgang mit wiederkehrenden Textsequenzen, "Suchen und Ersetzen" erlaubt die blitzschnelle Auswechslung bestimmter Textstellen. Der Assembler ist einer der schnellsten seiner Art: 50000 Textzeilen pro Minute – damit sind auch lange Quelltexte schnell umgewandelt. Erstellt werden können ausführbare Programme, Binär- oder Link-Files, die wie die Quelltexte wieder geladen werden können.

Der Debugger ist sicher in seiner Handhabung. Viele Funktionen erleichtern die Fehlersuche in Programmen, wie zum Beispiel "One Step", "Step n", "Run", "Jump", "Watch" und "Disassemble".

Der Monitor zeigt Ihre Programme "in Aktion": Disassembling, Ausgabe von Hex- oder ASCII-Dumps, Markierungen für Einsprungpunkte helfen, Fehler im Quelltext zu finden.

»ASM-One« bietet darüber hinaus noch weitere Spezialfunktionen: Über den Level-7-Interrupt läßt sich nach einem Absturz in 99 % aller Fälle noch etwas retten. Wird das entsprechende Signal über einen Schalter oder einen Taster an den Prozessorport geführt, springt der Rechner wieder in den Assembler zurück – Guru-Meditations verlieren ihren Schrecken. Darüber hinaus arbeitet »ASM_One« mit der »requester.library« zusammen.

Das Komplettpaket »ASM-One« ist nicht nur für Profis ein Muß, auch Einsteiger bekommen hier für ihre ersten Gehversuche das passende Werkzeug.

Systemvoraussetzungen:
Alle Commodore Amiga mit mindestens 512 KByte RAM



